1994

An extensible view system for supporting the integration and interoperation of heterogeneous, autonomous, and distributed database management systems

Cheng-Huang Yen
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9424278

An extensible view system for supporting the integration and interoperation of heterogeneous, autonomous, and distributed database management systems

Yen, Cheng-Huang, Ph.D.

Iowa State University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

An extensible view system for supporting the integration and interoperation of heterogeneous, autonomous, and distributed database management systems

by

Cheng-Huang Yen

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1994

DEDICATION

To my father, Wang-Deng Yen, and my mother, Chen-Yu Yen, for their love, support, and patience.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This dissertation would have been impossible without the support and encouragement of the members of my committee. I would like to thank my advisor, Professor Les Miller, for his insightful direction throughout my graduate career and his friendship that I value greatly. I also appreciate the support of Professors Gary Leavens and Johnny Wong. Their critical comments greatly improved the presentation of this dissertation.

I am thankful for the many friends that I am so lucky to have and I will miss for the rest of my life. I want to thank Mr. L. R. McFarlin for his guidance and support when I worked for the Residence Department; Professor Al Baker for his encouragement when I served as the coordinator of the SDG; Ms. Dorothy Lewis for listening and helping me when I worked for the Computation Center.

Finally, I want to thank my family. Thanks to my brother, Chun-Huei Yen, my sisters, Li-Ju Yen, Li-Hua Yen, and Li-Mei Yen, for always believing in me and encouraging me. Thanks to my son, Marty Shih-Jye Yen, for being my best friend in the past three years. And thanks to my wife, Sheue-Lih Chen Yen, for everything.

CHAPTER 1. INTRODUCTION

In this dissertation, we address the problem of integrating heterogeneous, autonomous and distributed database management systems (DBMSs). Today's organizations rely on a multitude of different computer and database systems to function normally and efficiently on a day-to-day basis. These existing systems have consumed significant capital investments although in most cases they have not been able to accommodate the increasing needs for information sharing on an intraorganizational or inter-organizational basis. Making revolutionary changes to existing organizational information systems would involve extensive re-training, re-investment, re-organization, re-engineering and possibly other problems at the organization-wide level. A favorable alternative is to use multidatabase systems that support the integration and interoperation of existing database systems while preserving their autonomy and functionality. The significance of multidatabase research is evidenced by the above facts and the numerous academic and industrial research projects.

The issues faced by multidatabase research are dynamic due to the changing technologies at both the software and hardware levels. This adds to the complexity of proposing a solid solution and developing a realistic multidatabase system. The multidatabase research that sparks Project *Zeus* aims at bringing the existing multidatabase technology to the next level by providing a unique solution that addresses

multidatabase issues at both the theoretical and system development levels. Our approach solves the modeling and system design problems that we will explain in Section 1.1.1.

This dissertation is organized as follows. The rest of this chapter presents the problem and covers the background material. Chapter 2 reviews related research in multidatabase areas. The theoretical foundation of our approach is presented in Chapter 3. Chapter 4 describes our design method and architectural design for a multidatabase system. In Chapter 5, we explain the exciting prospects of Project *Zeus* and point out our future research directions. For the remainder of this chapter, we describe the problems we are solving, the various terms used in this dissertation, how this research was originated, the scope of this research and our contributions.

1.1 Background

Database applications reflect the needs and functionality of organizational information systems. DBMSs provide the tools and mechanisms for the modeling, representation, and storage required by data and applications. Organizational computing and information systems can be viewed as a multidatabase community within which the functional hierarchies of organizations are embedded and supported. Before we introduce the problems we are solving and detail our solution, we provide definitions for a number of terms used throughout this thesis.

Multidatabase Community and Cell. A multidatabase community is the union of a set of applications, a set of database systems, and a set of computing systems:

$$\mathcal{MC} = \mathcal{MC}_{\mathcal{A}} \cup \mathcal{MC}_{\mathcal{D}} \cup \mathcal{MC}_{\mathcal{S}}, \text{ where}$$

$\mathcal{MC}$ denotes a multidatabase community, $\mathcal{MC}_{\mathcal{A}}$ denotes a set of applications, $\mathcal{MC}_{\mathcal{D}}$ represents a set of database systems, and $\mathcal{MC}_{\mathcal{S}}$ refers to a set of computing systems that host the elements in $\mathcal{MC}$. The scope of a multidatabase community may cross organizational or administrative boundaries. We refer to an administrative domain as a **cell**. Every part of $\mathcal{MC}$ is in some cell, but the cells may overlap. Each cell has its own semantics in terms of its modeling, requirements, and functionality.

Semantic Relativism. Semantic relativism ($\mathcal{SR}$) [22] refers to the capability of supporting multiple interpretations of the same stored data. For example, the data stored in a purchase order database may be displayed either ordered by the purchase order number or grouped by the departments that issued the purchase orders. In a distributed multimedia information system, semantic relativism can be interpreted as different ways of performing spatio-temporal composition of distributed multimedia objects [71]. The notion of semantic relativism is important in our research because we want to hide applications from knowledge of how to use the underlying DBMSs and at the same time allow different applications to have different views of the same data from different database systems.

Multidatabase System and Distributed System. A multidatabase system is a distributed system that functions as a front-end to multiple cooperating DBMSs. A distributed system is a system with processing elements and computer peripherals, connected by a network. In a client-server model, a distributed system can be described as a set of cooperating servers and clients. A multidatabase system is capable of managing persistent data and accessing data from several databases

managed by heterogeneous DBMSs in a distributed system. The databases involved are heterogeneous in the sense that they are implemented by different data models and are managed by different sets of operations. They are distributed in the sense that individual databases are under local control and are interconnected by computer networks.

Distributed Abstraction Modeling. In a distributed computing environment, we refer to distributed components such as the communications components[1], operating systems, hardware platforms, DBMSs, and other computing systems that interact with one another within the computing environment. Distributed abstraction modeling ($\mathcal{DAM}$) refers to the capability of protecting users from the added complexity of distribution and heterogeneity by providing different levels of transparency to the distribution and heterogeneity of distributed components. In our research, the notion of distributed abstraction modeling is important because we assume that the computing environment is distributed and our approach should be applied to a more global architecture in order to increase information sharing. We will give a definition for $\mathcal{DAM}$ in Section 3.2.

1.1.1 The Problems

In this research, we are solving the problems related to the modeling and the design of multidatabase systems. Figure 1.1 shows the problems we are solving and the proposed solution. Part 1 of Figure 1.1 identifies four cases that exemplify the ways the applications interact with the participating DBMSs. Case (1) shows that

[1] For example, a network server is a communications component.

an application exchanges data/messages with a DBMS. In this case, the application only needs to know how to communicate with the DBMS. Case (2) shows that an application sends the same data/messages to multiple DBMSs. Since each DBMS may have different ways of interacting with the outside world, we expect that there exists a translation to make the data/messages meaningful to the receiving DBMSs. Case (3) indicates that an application requires to access data from multiple DBMSs at the same time and Case (4) shows that these data must be presented to the application in different ways. In real-world applications, we can find many examples that correspond to the above cases. It is clear that in addition to the existing DBMSs there is a need to provide another layer of software system that protects applications from having to know how to interact with each kind of DBMSs. This is the major problem addressed by most multidatabase research. In our research, we address this problem in a more global architecture because global information sharing has become increasingly important. We also address related issues by assuming that the computing environments are characterized by increasing heterogeneity, distribution, and cooperation. For the remainder of this section, we explain the problems and the associated issues along two dimensions: modeling and system design.

Modeling. The modeling problem is to create a theoretical foundation that appropriately addresses the modeling and semantic issues related multidatabase integration. In other words, we need to model how the integration of data from multiple DBMSs can be represented and mapped to the requirements of applications in a distributed computing environment. For example, suppose that an application running at site A needs to bring in several image files from remote sites. The formats

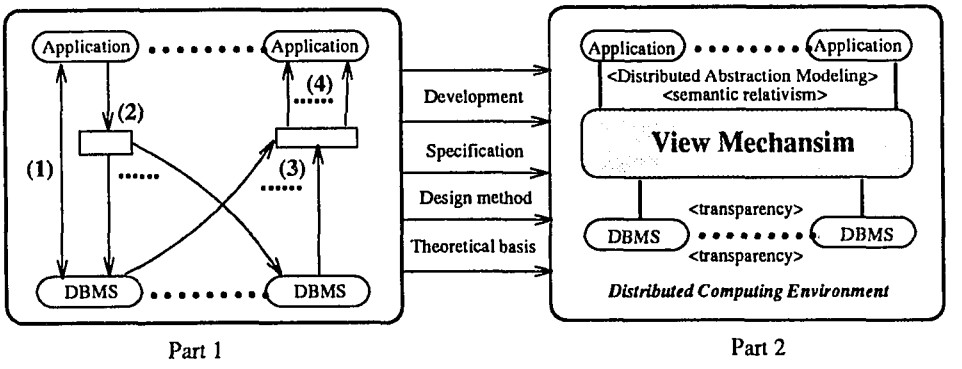

Figure 1.1: The problems we are solving and the proposed solution.

of these image files are all different. To relieve the application from dealing with the various formats, the multidatabase system must coordinate the conversion of these image files to the format displayable at site A. Since these image files are stored in remote sites, they have to be transported to site A over communications networks. Even after all the images files are converted and transported to site A, the application may integrate all the images into a text file or a single image that is meaningful in the context of the application domain. A multidatabase system cannot hardwire these solutions because whenever the multidatabase system needs to be refined or adapted to accommodate a new computing environment or a new DBMSs, the entire system design would have to be revised. This would reduce the extensibility of the system and hinder the possibility of applying the system to a more global architecture. We analyze two different aspects of this problem:

1. To provide data for applications in a meaningful way, the representations and semantics of data have to be mapped to the modeling, requirements, and functionality of applications. The description of such a mapping must be stable and

uniform. The description must be stable in the sense that it is independent of the representations and storage of the data involved. This is required in order to deal with large objects and to avoid unnecessary semantic conflicts. The description of such a mapping must be uniform in the sense that it is transparent to the heterogeneity of participating DBMSs and applications. This is required in order to keep the complexity of the system under control.

2. In addition to describing the mappings between the representations of data and applications, there are other details related to how the data are retrieved, converted, transported, and presented to the applications; i.e., how to support the mappings. We want to enforce a clean separation between these details and the syntax and semantics of the descriptions for the mappings. Such a separation is required because how the data are presented or used by the applications only depends on the semantics of the data in the associated DBMSs and the applications. This has nothing to do with the details of the mechanisms that package and ship the data from DBMSs to applications. Such mechanisms are different for different computing environments. Application developers should be protected from needing to know these mechanisms. From system developer's point of view, the separation also simplifies the system design.

System Design. To achieve global information sharing, we expect that a multidatabase system will be deployed in many different computing environments. The multidatabase community will be hosted by a more global architecture. We cannot afford to redo the system design for each kind of computing environement. We also want to avoid making significant changes to the system design upon additions of

new DBMSs or applications. Otherwise, the system development would be too complicated and the system maintenance would be unwieldy. We need a system design method that achieves the following design criteria: portability, scalability, interoperability and extensibility. **Portability** ensures that the system design and even the source of the system can be used in different computing environments without significant change. **Scalability** means that the increase in the number of participating sites will not have major impact on the system design. **Interoperability** refers to the capability of multiple multidatabase systems to interoperate with each other and with the computing environment. Finally, the system must be **extensible** to accommodate the heterogeneity of new cooperating DBMSs and evolving computing environments without corrupting other design criteria.

1.1.2 The Proposed Solution

Figure 1.1 gives an overview of our proposed solution. The problem related to **modeling** should be addressed by a rigorous theoretical foundation that can later be expanded and be used to verify whether the modeling issues are properly dealt with in the system design. Our research was motivated by the idea of partial integration of multiple DBMSs. The view concept was introduced into our approach for two reasons. First, there exists an analogy between *integrating multiple DBMSs* and *constructing views from existing views.* Second, The notion of views is the most natural way to implement semantic relativism[22]. We use the layering technique to simplify the integration model which forms the core of our view mechanism, the *Zeus View Mechanism* ($\mathcal{ZVM}$). The layered architecture also separates the mappings of the semantics of data from the details of how to ship the data. The $\mathcal{ZVM}$

hides applications from the details of how to use the underlying DBMSs and at the same time allows different applications to have different views of the same data from different database systems. In other words, the $\mathcal{ZVM}$ create a uniform interface for applications to access multiple database systems as though there were a single information source. The $\mathcal{ZVM}$ supports semantic relativism and distributed abstraction modeling, and provides a unique solution to the modeling problem.

The design of a multidatabase system based on the $\mathcal{ZVM}$ creates the *Zeus View System* ($\mathcal{ZVS}$). We have developed a framework-based design method to deal with the **system design problem** through large-scale reuse. Our idea is based on the fact that the system can be developed via reusing *frameworks* of plug-compatible software components. The high-level design of frameworks is portable across hardware platforms. Therefore, we can reuse the system design when the system needs to be developed for a new computing environment or when new DBMSs are added to the multidatabase community. Now that we have described the conceptual structure of our solution, we can define the scope of our research that turns our concept into a working multidatabase system.

1.2 Scope

We have identified the following components as integral to the development of the *Zeus View System*: a theoretical foundation, a design method, and a distributed object infrastructure. The theoretical foundation provides a rigorous basis for the $\mathcal{ZVM}$. We have developed an **object model**, a **mapping methodology**[2], and an **integration model** to formalize the $\mathcal{ZVM}$. We have combined object-oriented

[2]We will explain what a mapping methodology is in Section 3.4.2.

techniques and software engineering concepts to create a **design method** based on the use of frameworks. To provide a distributed object infrastructure, we integrate the CORBA specification[122] into our system design and development. A complete high-level design specification of our multidatabase system written in CORBA IDL has been included in Appendix B. We have also defined a **computation model** based on the mappings between views and CORBA objects. Our research has been focused on the above components. Although we have not started to deal with the issues related to the multidatabase transaction management, the completion of the $\mathcal{ZVS}$ will provide a test bed for us to explore those issues in a more pragmatic manner. The implementation of the $\mathcal{ZVS}$ is beyond the scope of this dissertation. The perspectives of our future work and research directions are described in Chapter 5.

1.3 Contributions

The major contribution of this research lies in the creative concept and design of a new **view mechanism** and a **view system** that has a broad applicability to future integration and interoperation technologies. Our experience with Project *Zeus* has also addressed a number of key issues and allowed us to propose solutions that have not been extensively discussed or explored in other existing work.

Our direction in multidatabase system design is motivated by the micro-kernel architecture[50] in operating system design, and the Open OODB[111] in object-oriented DBMS design. The common goal of these approaches is to achieve an extensible, scalable, portable, and interoperable design. In Project *Zeus*, the issues are even more complicated due to the increasing scale, heterogeneity, distribution, and

cooperation of the participating systems and surrounding computing environment. Our solution to these issues results in another important aspect of our contribution reflected in a complete **system development method** that has the attributes not found in existing multidatabase systems; i.e., a rigorous formal foundation, strong structuring concepts, extensibility, transparent and high-level reusability of system components, portability over full range of machine architectures, support for scalability, support for dynamic configuration, enablers for transparent distribution, and conformance with Open System standards. The system development method is based on the application of an object-oriented and framework-based design to the development of large-scale distributed systems. We need such a system development method in order to make it possible for the $\mathcal{ZVS}$ to be implementable, maintainable, and useful in a global architecture.

Up to this point, not only have we provided a unique solution to multidatabase issues and multidatabase system development, we have also established a solid foundation that will provide an open framework for composing, coordinating and configuring distributed applications which in turn support transparent sharing of global resources.

CHAPTER 2. RELATED RESEARCH

Multidatabase research has been quite intensive and extensive in recent years. A taxonomy of multidatabase systems is described in [21]. A collection of articles on recent advances in multidatabase research can be found in [57]. Related work in multidatabase research can be classified into a number of categories: fundamental research, database interoperability, object-relational interface, multidatabase system development, and standardization activities. We will give a survey for each of these categories in this chapter. Before we start, it is worthwhile to point out that different approaches may result in different degrees of integration or different levels of interoperation. Integration and interoperation technologies are required by a spectrum of applications and there is no single approach or system that can be applied to all applications. However, the experience and the technical merits or flaws in the various approaches of exisiting multidatabase research have given us valuable guidance in the development of our solution.

2.1 Fundamental Research

The fundamental research in multidatabase includes modeling issues, mapping methodologies, and semantic issues. The **modeling issues** center around the creation of a common data model for modeling the integration and interoperation of

multiple database systems. Saltor, et al., discussed the suitability of data models as canonical models for federated databases [92]. The representation ability of a model was discussed based on expressiveness and semantic relativism. Barsalou and Gangopadhyay proposed an extensible metalevel system in which the syntax and the semantics of data models, schemas, and databases can be uniformally represented [11]. They named this system M(DM). It is also referred to as an open framework for interoperation of multimodel multidatabase systems.

The **mapping methodologies** refer to the mappings of participating database systems to the semantics of client applications as well as the syntax and semantics of the constructs for describing such mappings. Konstantas described the mapping methodology in the context of type relations, type matching, and object mapping [69]. Su, et al., proposed an object-oriented rule-based approach to provide a common data model and a mechanism for schema translation [105] that maps a local database schema to a global schema.

Semantic issues in multidatabase research were addressed in [96]. The integration model of a multidatabase system must be able to describe the semantics of data, metadata, integration, DBMSs, and applications. Semantic interoperability should be achievable in the integration model. Schema translation and schema integration must be semantic-preserving. Semantic relationships represents multidatabase interdependencies. The semantic relationships of both data and behavior need to be identified. Semantic heterogeneity refers to the disagreement about the meaning, interpretation, or intended use of the same or related data. Semantic heterogeneity must be identified and resolved among participating DBMSs. A multidatabase system also needs to support semantic reconciliation and semantic relativism. Semantic

reconciliation resolves semantic heterogeneity and identifies semantic discrepancy.

In our research, we also chose to use a common data model in order to reduce the number of mappings of the data models between DBMSs and applications. Our data model is based on an extensible object model. The object model is adjustable to different application domains. Therefore, we can use the same model with compatible extensions to describe the view mechanism as well as the design of the view system. This simplifies the modeling issues. Although it is hard to make a direct comparison between different data models proposed for multidatabase systems, we see our data model as a good choice for its uniformity and extensibility. Our view of the mapping methodologies follows that of Konstantas [69]. Our approach is based on a layered architecture that allows us to use the layering technique to encapsulate the issues related to the shipment of data in the lower layers of the layered architecture. The upper layers of the layered architecture can then focus on the mappings to application semantics transparent to the heterogeneity and distribution of the computing environment. This is how we are able to support distributed abstraction modeling and separate the semantics of a global request from the mechanism that actually executes the request. Semantic issues are being explored in our research. We plan to deal with semantic issues in the view constructs which are described in Section 3.4. The use of the layering technique to solve the modeling problem allows us to achieve different degrees of integration and different levels of interoperation. Other approaches have not been able to provide this kind of flexibility [11, 96, 105].

2.2 Database Interoperability

A shortcut to achieving information sharing is database interoperability. Database Interoperability is referred to as the capability of DBMSs to retrieve data and request services by invoking functions from each other. Recent architectures and approaches for database interoperability have been described in [124]. There are three basic approaches for accomplishing the connectivity required by database interoperability: database gateways, database drivers, and connectivity software that routes SQL. These three basic approaches spawn five major architectures for database interoperability. In general, the solutions to database interoperability are useful to a stable organizational information system that does not expect drastic change in application requirements and information structures. For multidatabase applications in a global architecture, database interoperability is not as useful because the number of participating DBMSs is very large and it is not feasible to provide a proprietary interface for each pair of DBMSs. We will briefly describe these architectures and then explain why they do not provide the solution to the problems we are solving.

PC Front Ends With Database APIs. This architecture tries to solve the problem of allowing applications to access a variety of back-end data sources through a uniform interface, which in this case refers to an application program interface (API). Examples include Microsoft's Open Database Connectivity, Borland's Open Database API, and the SQL Access Group's (SAG's) Call Level Interface. However, these APIs lack the support for the setup and management of communication sessions, coordination of operations in multiple DBMSs, and query optimizations which are all required in handling a global request in a multidatabase system.

Relational DBMSs With Conventional Gateways. The role of conventional gateways for RDBMSs is to translate information and structure of foreign data sources into the host database's format. This is functionally similar to database drivers. The conventional gateways provide the resources needed for distributed database access in the server DBMS, and a stable and transparent architecture for database interoperability. Examples of conventional gateways come from Ingres, Informix, Oracle, etc. Adding conventional gateways to RDBMSs does not provide a total solution to multidatabase issues since it only supports RDBMSs.

Open Gateways. Open gateways provide the functions required to transparently connect clients to remote data sources and to route requests for operations on remote databases to the proper target. Users of open gateways can implement their database interoperability software independently of their DBMS software. Most open gateways also provide APIs for building client applications and new gateways. Examples of open gateways include the products from Sybase, Information Builder, and Software Publishing Corp. Although the name of this architecture implies an open architecture, the implementations are biased toward existing RDBMSs. The concept of open gateways is similar to the choice of a common data model in a multidatabase system.

Database Connectivity Software. Database connectivity softwares take SQL statements from client applications and deliver them via the proper networking protocol to the target database. Performing this function often requires transformations of the transport protocol. Database connectivity softwares only address networking between applications and databases which is part of a total database

interoperability problem. Examples include Database Gateway from Micro Decisionware, SequeLink from Techgnosis International Inc., and DAM from Apple Computer Inc. Data connectivity softwares do not provide a total solution to multidatabase issues because only RDBMSs are supported.

Database Encapsulation Software. Database encapsulation softwares encapsulate target DBMSs by objects that represent their functions and access methods. Users are shielded from the differences in types of databases. The database encapsulation approach comes closest to supporting all of the requirements of database interoperability. Examples of database encapsulation software include the products from Constellation and OpenBooks Inc. Database encapsulation softwares introduce the notion of database encapsulation. However, database encapsulation software has not been able to further encapsulate the heterogeneity and distribution of the computing environment. This hinders the possibility of using database encapsulation software in a global multidatabase environment.

2.3 Object-Relational Interface

There are two major aspects of an object-relational interface. First of all, an object-relational interface can be viewed as an interface to bridge Relational DBMSs (RDBMSs) and Object DBMSs (ODBMSs) such that users of one DBMS are allowed to access the data stored in either format. Secondly, an object-relational interface can be viewed as a way to introduce object-oriented technologies to the applications associated with RDBMSs. Existing attempts to developing object-relational interfaces have both or either one of the above two aspects. We give a review of a number of

object-relational interfaces to illustrate the work in this area. In general, an object-relational interface is quite useful in the real world since many existing DBMSs are either RDBMSs or ODBMSs and many applications do require an interface between RDBMSs and ODBMSs. However, applying an object-relational interface to a more global architecture is not feasible because the interface has to be specialized for each pair of DBMSs based on the implementation-dependent features of each DBMS. When the number of DBMSs is large, this task would be unwieldy.

OR Interface. The OR Interface is a prototype developed at Iowa State University [90] in an environment where both a RDBMS and an ODBMS are used and users of one DBMS are allowed to access the data stored in either format. UNIX-based workstations and the AS/400 were chosen as the prototyping platforms. Although a more homogeneous environment could have been used or even having two different DBMSs on the same platform, that may however impede the effort of exploring some interesting heterogeneity and interoperability issues. UNIX-based platforms were chosen because of its widespread usage for desktop computing. The AS/400 was chosen because of its existence in large numbers in the commercial data processing environment. In the OR Interface system, a request for data may be initiated by an interactive query language, an application program using an embedded query, or an object programming language requesting a persistent object. Users of one DBMS are shielded from knowing the details of another DBMS while requesting data from either DBMS. The experience with the OR Interface has provided us with the details on how to create a non-intrusive interface[1] with a DBMS which is required in a multidatabase

[1] A non-intrusive interface allows a DBMS to retain its autonomy; e.g. the control of local resources or the right to stop to cooperate in the interface.

system. However, the OR Interface does not provide a complete solution to our problems because it is only useful for bridging ODBMSs with RDBMSs.

Object-Oriented Relational Database. Premeriani, et al., proposed an approach to combine an RDBMS with an object-oriented programming language to generate an ODBMS [87]. The basic idea of their approach is to buffer the database with an object-oriented layer that keeps relevant data in memory. Locking and update functions are built into the object-oriented layer. The object-oriented layer hides the database from applications. Application programmers can use object-oriented languages to create, save and restore persistent objects without knowing any operations provided by the target database. The notion of encapsulating databases via an object-oriented layer is useful and relevant to our approach. We have extended this notion to a layered architecture that provides more than one levels of encapsulation.

2.4 Multidatabase System Development

The focus of the research in multidatabase system development is to develop multidatabase systems that allow users to transparently access heterogeneous, autonomous and distributed database systems. There are two categories of issues related to the development of multidatabase systems. First of all, the functionality of multidatabase systems must be well-defined. For example, suppose a multidatabase system provides a uniform interface for applications. What is included in the interface and how the interface is tied to other components of the system need to be clearly specified. Secondly, the multidatabase system must be made feasible under all pragmatic considerations. For example, a multidatabase system must be extensi-

ble, scalable, portable, and interoperable with other systems. How the performance can be optimized based on the configuration and application requirements should be addressed. Although the same functionality can be achieved by different designs and implementations, failure to accommodate pragmatic considerations will render the multidatabase system infeasible or even useless. In the remainder of this section, we review existing approaches to multidatabase system development. Under each approach, we describe various prototypes and projects that are based on the approach. We also explain why the problems we addressed in Chapter 1 cannot be solved by these approaches.

2.4.1 Global Schema Approach

This approach creates a global schema from local external schemas. The global system supports a common data model and a global data language. Multidatabase users view the global schema as the definition of a single database. The heterogeneity of local DBMSs is hidden from users. The global schema approach requires a total integration at the schema level. Maintaining a global schema becomes unwieldy when the addition and update of participating DBMSs are frequent. The problem is even worse when we apply this approach to a global architecture. Moreover, different applications require different degrees of integration. Enforcing a total integration on all applications is too restrictive.

- **Multibase.** Multibase is a software system for supporting integrated access to pre-existing, distributed heterogeneous databases [40]. The system provides users with a unified global schema and a global high-level query language. Local database schemas are translated to local schemas modeled by the global

data model. The global schema is created from a set of local schemas and an integration schema. The integration schema maintains the mappings between local schemas and the global schema. The global schema allows users to pose queries against what appears to be a homogeneous and integrated database. The major weakness of Multibase is that even for the database whose data will not be used at all by any application, its database schema still needs to be integrated into the unified global schema.

- **DATAPLEX.** DATAPLEX is a heterogeneous distributed DBMS developed at General Motors Research Laboratories [34]. The key concept in DATAPLEX is to use the relational model as a common data model and the Structured Query Language (SQL) as a global query language. The architecture of DATAPLEX is extensible to any DBMS and file system. The relational model is insufficient for modeling complex objects that appear in ODBMSs. Using the relational model as a common data model fails to address the modeling problem in a multidatabase environment.

2.4.2 Multidatabase Language Approach

This approach puts most of the integration responsibility on users. Global users are aware of multiple data sources. Instead of providing a global schema, a common name space is defined across all participating DBMSs. Users use the global multidatabase language to define the sources of data, and how the data is integrated, transferred and presented. An example of the multidatabase language approach is SWIFT [108]. The SWIFT system was developed for cooperative fund transfer applications. A common communications architecture and a set of financial transaction

messages are used as a common basis for interactions among participating partners in the SWIFT system. The multidatabase language approach puts too much responsibility on users. One of the problems we are solving is how to make multidatabase users transparent to the heterogeneity and distribution of the computing environment and DBMSs. The multidatabase language approach fails to provide this kind of transparency.

2.4.3 Federated Database Approach

In a federated database, each local DBMS maintains a partial global schema that contains only the global information used by local users. There is no single global schema. The local site needs to work closely with a set of inter-related sites to set up the partial global schema. The set of inter-related sites forms a federation. A federation consists of several sites and a single federal dictionary. Each local site in the federation controls its interactions with other components by means of an export schema and an import schema. An extensive definition of federated databases can be found in [54]. A survey of existing federated database systems has appeared in [98]. Federated databases have tried to avoid the shortcomings of the global schema approach and the multidatabase language approach. The notions of an export schema and an import schema have proven to be quite useful in our approach. However, existing work in federated database has not addressed how the federations interact with one another to achieve a more global information sharing.

2.4.4 Object-Oriented Approach

In recent years, object-oriented approaches to multidatabase system development have started to emerge to utilize the object-oriented modeling power and object-oriented techniques [24]. These approaches are object-oriented in the sense that object-oriented models, properties, and techniques are used to assist in the development of multidatabase systems at different levels. For example, the notion of objects can be used to encapsulate the functionality of DBMSs through which multiple levels of abstraction can be achieved and the underlying heterogeneity can be hidden from users. Object-oriented techniques may also affect the architecture of a multidatabase system. For example, a multidatabase system may rely on a distributed object infrastructure for object transfer and remote method invocation. A survey of multidatabase research and systems using object-oriented approaches has appeared in [24]. In general, object-oriented approaches have many advantages over other approaches. Although it is hard to give a direct comparison between our approach and other object-oriented approaches, it is clear that none of existing approaches have addressed the system design problem. To deploy a multidatabase system in a global architecture, the system design must be extensible, portable, scalable, and maintainable. We address this problem by providing a framework-based design approach that applies object-oriented techniques to achieve code/design reuse. This is how we are able to reduce the complexity of developing a multidatabase system in a global architecture.

- **Pegasus.** The Pegasus Heterogeneous Multidatabase System [4] has a common data model and a global data language. Each local DBMS exports a local schema which is then mapped to an import schema in the multidatabase

system. The integrated schema is composed of import schemas and other integrated schemas. Pegasus does not support encapsulation although it does support an object model. In Pegasus, an object has a type and can be operated on by a set of functions defined separately from the object. Recent advances in Pegasus adopted the notion of abstract view objects to facilitate the integration of multiple object DBMSs [32]. A set-theoretic foundation is provided for dealing with object identification, object integration and function inheritance in the context of abstract view objects. In contrast to Pegasus' approach, our view mechanism deals with integration and interoperation semantics through type matching, type relations and object mapping. This allows us to provide a much simpler solution to the modeling problem by using a single construct; i.e., *Zeus* views, for modeling integration and interoperation semantics. Our view mechanism is further integrated with CORBA [122] through a two-step mapping that attaches runtime semantics to *Zeus* views.

- **ViewSystem.** ViewSystem is developed at GMD-IPSI as an object-oriented programming environment to provide uniform access to heterogeneous information bases such as databases and file systems [60]. ViewSystem provides an object-oriented query language and a method language. In ViewSystem, information systems are modeled by the VODAK data model. The mappings between schemas of participating information systems and VODAK are handled by schema transformations. Integration operators and class constructors are provided to facilitate both data and semantic integration. ViewSystem has applied object-oriented techniques to modeling issues. Although ViewSystem has the potential of being extended to work in a global architecture since the

participating information systems are not restricted to DBMSs, the system design problem was not addressed.

2.4.5 Miscellany

- **Superdatabase.** The notion of superdatabases [88] was developed at Columbia University for investigating transaction processing in multidatabase environments. A superdatabase is intended to glue component transaction processing systems in a hierarchy. Participating database systems export external schemas to describe local data in a global data model. A superdatabase is composed of local external schemas and other related superdatabases. The hierarchy of the composition of superdatabases matches the way the multidatabase transactions are integrated and structured. A declarative interface is provided to access the composed superdatabase. This approach only addresses the problem of multidatabase transaction management. It does not provide solutions to modeling and system design problems.

- **Carnot.** Carnot is a system for integrating information resources using a large knowledge base [37]. In Carnot, the global schema approach was adopted. However, the implementation of Carnot has avoided shortingcomings of the global schema approach in three ways. First of all, the global schema does not need to be re-generated each time a new resource is added. This is done through the use of the Cyc knowledge base. Secondly, in addition to structural descriptions, Carnot records schema knowledge, resource knowledge, and organization knowledge to aid schema integration and resolve semantic differences. Thirdly, Carnot uses a set of articulation axioms to handle the mappings between infor-

mation resources and the global schema. Adopting the global schema approach eliminates the possibility of using Carnot in a more global architecture.

2.5 Standardization Activities

It should be noted that standardization activities also contribute to the advances of integration and interoperation technologies. For example, Remote Database Access (RDA) is a standard drafted by the International Organization for Standardization (ISO) [125]. RDA uses the Open Systems Interconnection services as the basis for RDA services. With the implementation of this standard, users will have a single, well-defined interface for heterogeneous environments. SQL Access is another standard drafted by the ISO [126]. Many vendors have started to implement SQL servers which conform to this standard to increase database interoperability. The standardization of object DBMSs and object query languages will also affect the multidatabase research. We will elaborate more on this in Chapter 5 and Appendix A. The standardization of distributed object services and architectures has significant impact on those multidatabase systems built on object-oriented approaches. Basically, the transport of objects and remote method invocation will be much easier and reliable as long as the involved parties are compliant with the standards for distributed object services and architectures. The components of a multidatabase system and its associated environment are too extensive to be covered by any existing standards. It is also true that standards are not always acceptable by the majority. However, the standards do provide convenience when it is widely accepted and can be used in part of the multidatabase system. In Chapters 3 and 4, we will explain how we accommodate recent standard specifications in our solution.

CHAPTER 3. THE ZEUS VIEW MECHANISM

In this chapter, we introduce the foundation of our approach to integration and interoperation technologies, the *Zeus View Mechanism* ($\mathcal{ZVM}$). We use a single construct, the *Zeus* view, as our basic modeling unit. The *Zeus* view does not have a stored state nor does it provide any global resources. It is only used to describe available services and resources such that the information can be recalled by the $\mathcal{ZVS}$ to coordinate the sharing of services and resources. Different types of *Zeus* views along with the components of the surrounding computing environment form a layered architecture that provides multiple levels of abstraction and encapsulation for the $\mathcal{ZVM}$. For the remainder of this chapter we start with an introduction of the view concept. Then we describe how the *Zeus* views lead to the layered architecture. Finally, we provide a formal description for the theoretical foundation of the $\mathcal{ZVM}$. This theoretical foundation is composed of an extensible object model, a mapping methodology, an integration model, and part of a computation model.

3.1 Evolution of the View Concept

Due to the diversity of existing view concepts, it is hard to make a precise comparison between the various views used in the same or different areas of research. However, we would like to explain the idea of how view concepts have been used in

different application domains and why we chose views as the basis of the $\mathcal{ZVS}$. We start with a review of conventional and recent use of views to give a general notion of views, although not all uses of views are directly related our research.

Conventional Views. The conventional notion of views is often used to describe derived data in order to achieve better protection and flexible access to shared information. For example, relational views can be represented by non-procedural queries to describe derived data. Views have also been used in software engineering. For example, views in [48] were used to let multiple tools share a common object base.

View Objects. In recent years, the view concept has been extended to utilize object-oriented techniques. This extension has been applied to a variety of application domains. For example, the notion of view objects was introduced in [113] to integrate the abstraction capabilities of programming languages and the conventional view concept of database systems. In [87], an Object-Oriented Relational DBMS was implemented and applied to a spectrum of applications. The view concept in [99] is used to create an architectural building block for object-based software environments. In [12], object-based views are used to update relational databases. The issues of view update semantics are discussed and investigated in [33, 94]. Numerous approaches have also emerged to encapsulate information and software systems by views. For example, abstraction and view mapping capabilities are proposed in [53] to support federation of heterogeneous software and databases. A sophisticated view mechanism is introduced in [1] to help restructure data or integrate databases. Object-oriented views are used to integrate heterogeneous information systems in [38, 60].

Views in Project Zeus. The word, view, means what one can see from where one is. We introduced the view concept into our approach because views possess a number of properties. First of all, views are combinative which implies that we can use views as the basic unit for integration. Second, views are flexible. Different views created from the same set of data may be tailored to different applications. The notion of views is the most natural way to implement semantic relativism [22]. Third, views are declarative. We can use views to hide heterogeneity and implementation details. The *Zeus* view has a number of characteristics that separates our approach from previous research. First of all, the *Zeus* view has broad scope. It is capable of modeling a wide range of application domains, and applications may span across organizational and administrative boundaries. Second, the interaction between *Zeus* views is characterized by specification level interoperability [115]. The composition of *Zeus* views is easy and flexible due to the separation of specifications and implementations. Third, the *Zeus* views carry meta information [76] and we model the *Zeus* views by a single basic structure. The incorporation of the metadata in the *Zeus* view improves global information search, security control and transaction management in the $\mathcal{ZVS}$. Modeling the *Zeus* view via a single basic structure facilitates the integration of views.

3.2 Layered Architecture

The $\mathcal{ZVM}$ has a layered architecture shown in Figure 3.1. Layering is a widely accepted structuring technique which decomposes one problem into a number of more manageable subproblems. The concealment of the functionality of lower layers from upper layers forms the basis of multiple levels of encapsulation and abstraction. The

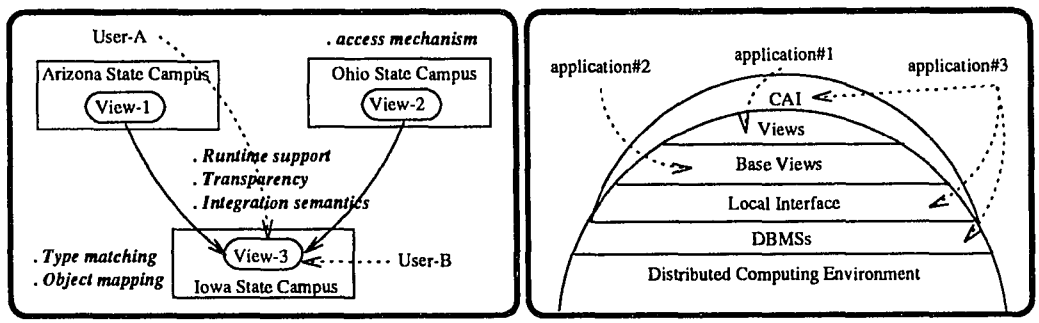

Figure 3.1: Wide-area abstraction and layered architecture.

layered architecture of the $\mathcal{ZVS}$ has six layers; i.e., layer 1 (Distributed Computing Environment), layer 2 (DBMSs), layer 3 (Local Interfaces), layer 4 (Base Views), layer 5 (Views), and layer 6 (Common Access Interface). The major role played by the layered architecture is to support semantic relativism and distributed abstraction modeling which are solutions to the modeling problem. The key to the creation of the layered architecture is centered around the notions of wide-area distributed abstraction and multi-level encapsulation and abstraction. We explain these important concepts next.

Wide-Area Distributed Abstraction. Figure 3.1 gives an example that illustrates the complexity of the issues in Project *Zeus*. We assume that the need for integration arises within and between each campus computing environment. In other words, the integration has a more global context. In Figure 3.1, View-3 is constructed from View-1 and View-2. Both user-A and user-B access global information through View-3. First of all, View-1 and View-2 must carry enough information to access data from the local DBMSs. View-3 will guide the multidatabase system to perform the integration of data from View-1 and View-2. The integration must reflect

the semantics of real-world applications. Based on the request, data objects will be moved between hosts. Migrating an object from one host to another means moving both data and operations of the object. The remote host may have the same data representation and object types. If not, object transformations and type translations should be provided to instantiate a corresponding type in the local runtime system. The semantics of the object should be preserved after the migration. Furthermore, the multidatabase system may need some sort of intelligence to optimize the performance. For example, the data from View-1 might be directly made available to user-A without network communication. The goal of the multidatabase system is to make all these issues transparent to applications. We refer to this kind of transparency as **wide-area distributed abstraction.**

Multi-level Encapsulation and Abstraction. Layering embedded in the layered architecture results in multiple levels of abstraction and encapsulation. In the first level of encapsulation, the interaction between the $\mathcal{ZVS}$ and the local computing environment is defined in a portable interface. In our case, this interface is defined in CORBA/IDL [122]. In Chapter 4, we will elaborate on what CORBA/IDL is and how it is used in Project *Zeus*. In the second level of encapsulation, the $\mathcal{ZVS}$ uses the schema translation to map local database schema to local interfaces which are modeled by an extensible object model (EOM)[1]. The mappings between these local interfaces and local schemas enable the $\mathcal{ZVS}$ to invoke local access routines on behalf of a global request. The heterogeneity of participating DBMSs and transport mechanisms of remote data access are hidden from upper layers. In the third level of

[1] We will elaborate on the EOM in Section 3.4.1.

encapsulation, base views are derived from local interfaces to describe the data that the local DBMS is willing to import or export. The import/export mechanisms are hidden from the upper layers. In the fourth level of encapsulation, base views and existing views can be used as building blocks for constructing new views that are tailored to the needs of different applications. The view construct hides the integration and composition details from its upper layer. In the fifth level of encapsulation, the Common Access Interface (CAI) determines how views are seen from the outside world. The design of the CAI shields applications from needing to know the internal structures of views and makes all participating DBMSs appear as a single source of services and resources. The *Zeus* view constructs; i.e., Local Interface, Base View, and View, sitting in layers 3, 4, and 5, respectively, provide multiple levels of abstraction. The notions of multi-level encapsulation and abstraction can also be summarized as a mapping between a set of information sources and an application

Summary. Figure 3.2 summarizes the role of the layered architecture in our research. We assume that the multidatabase system will be used in a global multidatabase community hosted by a distributed computing environment. In Figure 3.2, we use different shapes to indicate that participating DBMSs may have different external interfaces and different internal representations of data. Similarly, multidatabase applications have different data requirements and different interpretations of received data. It is infeasible to provide an interface for each pair of a DBMS and an application. We want to use a single software layer to protect applications from the heterogeneity and distribution of DBMSs. All the participating DBMSs should appear as though there is only a single DBMS. We use the layering technique

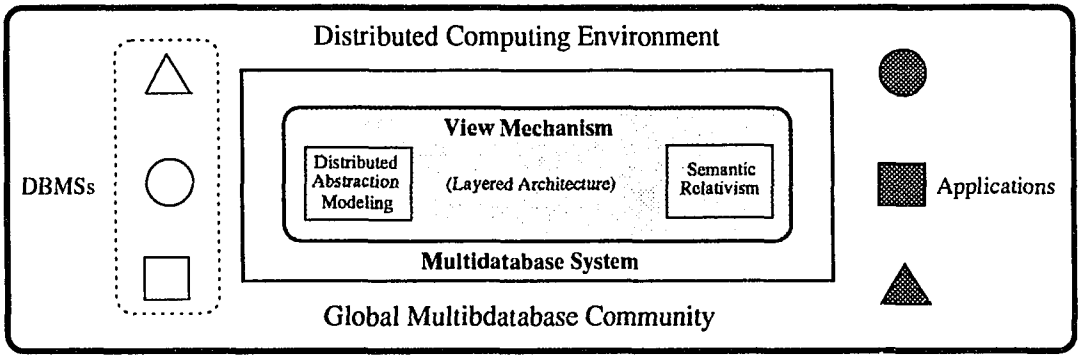

Figure 3.2: The view system and the view mechanism.

to deal with the complexity of this problem. The layered architecture is the core of the $\mathcal{ZVM}$. The lower layers of the layered architecture hide the heterogeneity and distribution of DBMSs and other distributed components from upper layers of the layered architecture. This provides a basis for distributed abstraction modeling. Therefore, upper layers of the layered architecture can concentrate on modeling application requirements and provide different views of a large amount of shared data. This is how semantic relativism is supported. Our multidatabase system implements the $\mathcal{ZVM}$.

3.2.1 A Comparison of Terms

Figure 3.3 shows how some commonly used terms in multidatabase research are compared to the terms used in the ANSI/ISO 3-schema architecture. The ANSI/ISO 3-schema architecture separates the conceptual schema of an application from its external schema and physical schema. This separation enhances the design portability of an application across different computing environments since the application

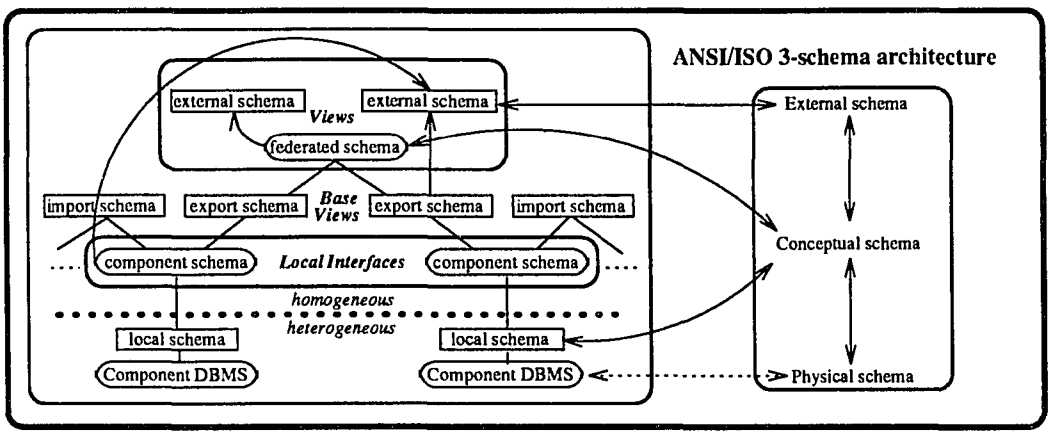

Figure 3.3: A comparison of terms.

can change its design by changing its conceptual schema. The external schema and physical schema of the application in different computing environments will be automatically adjusted upon the occurrence of changes in the associated conceptual schema. In Figure 3.3, all the constructs above the dotted line are represented by a homogeneous global data model. The heterogeneity is confined to the space below the dotted line. The local schema is mapped to the component schema or local interface through a semantic-preserving schema translation. The export schema is derived from the component schema and describes the data that the local DBMS is willing to share with a limited set of clients. An import schema describes how the external data can be imported into the local DBMS. In the $\mathcal{ZVS}$, base views correspond to export schemas and import schemas. The $\mathcal{ZVS}$ implements both partial integration and total integration. We refer to all other constructed views as $\mathcal{Z}eus$ views which correspond to the federated schema and the external schema.

3.3 A Running Example

We use a running example in this thesis to illustrate how the $\mathcal{ZVS}$ can be used for multidatabase integration and interoperation. The descriptions of the real-world entities and modeling constructs in this example are informal. What we would like to motivate is how the system architecture of the $\mathcal{ZVS}$ achieves the application requirements. The databases used in our example are defined in Figure 3.4.

Environment and Database Schemas. We assume that an airline maintains a relational database for the reservation and flight information. The airline also offers a MileageAward program that gives customers free flights based on their accumulated flight mileage. The MileageAward program maintains its information in an object database. The airline operates a world-wide distributed computing environment. The MileageAward program database has two class definitions; i.e., MileageAwardMember and Flight. Flight is an abstract class and has no instances. The reservation information is kept in a reservation relation. The flight information is kept in a flight-info relation. Both relations are hosted by a relational DBMS.

Application Semantics. There are three applications that require integrated access to both relational and object databases in our example. In application#1, the MileageAward program prints out a mileage balance report for each member every year. This application will need the flight information from the relational database to compute the total mileage for each member in the object database. In application#2, the members of the MileageAward program use the member card to update their flight record whenever they check in for a flight. The application

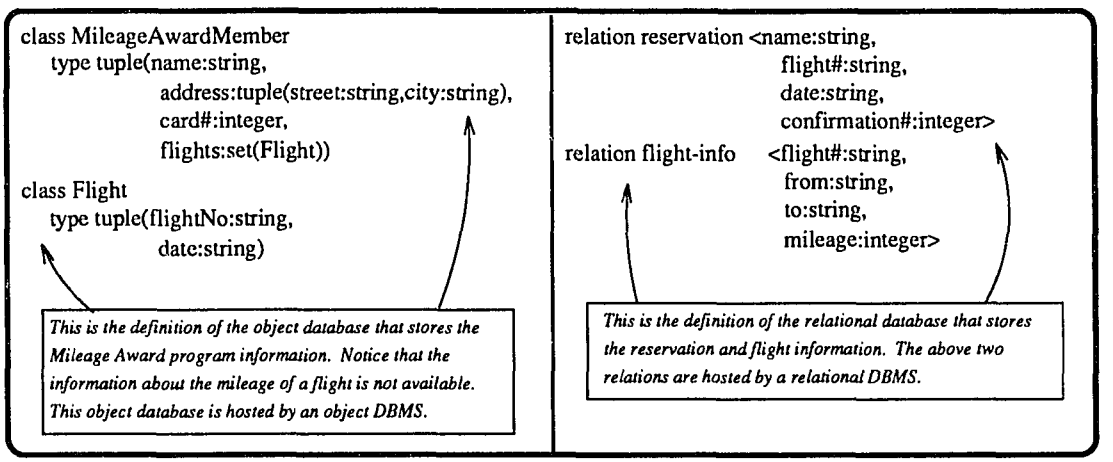

Figure 3.4: A running example.

requires the flight information to be sent from the relational database to the object database. In application#3, the airline provides a graphical user interface and a voice response service. The airline customers can confirm their reservations, check their flight records, or query the flight information via a phone call. The operators can use the graphical user interface to answer customer queries. Notice that we allow applications to access the $\mathcal{ZVS}$ at different levels. For example, application#3 can access the DBMSs through the CAI, the local interface, or the DBMSs.

3.4 Foundation

In this section, we provide a formal description for the theoretical foundation of the $\mathcal{ZVM}$. We start with the definition of an extensible object model (EOM). Based on the EOM, we describe our view of the real-world entities within a multidatabase community. The mappings we discussed in Chapters 1 and 3 are detailed in our description of the mapping methodology. This issue has been listed as one of the

key issues in multidatabase research [104]. The integration model of the $\mathcal{ZVM}$ provides the details of the integration mechanism that supports the integration of the view constructs. Finally, we use the OR model as an example to illustrate how the computation model of an actual multidatabase environment can be described in terms of the theoretical foundation of the $\mathcal{ZVM}$.

3.4.1 Object Model

We define an extensible object model (EOM) to provide a common framework for modeling real-world and conceptual entities in a variety of application domains. The idea of using a common data model for integration also appeared in [11, 105]. We adopt the terminology used in [123] to present the EOM. The EOM, as depicted in Figure 3.5, is composed of a core object model and a set of components. A component is a compatible extension of the core object model. A profile is a set of selected components. A profile along with the core object model provide the modeling facility for a particular application domain. The EOM is extensible in the sense that components can be added and then grouped by profiles to tailor to the requirements of existing and future application domains. The proposed core object model is influenced by on-going standardization efforts [45, 122, 123] and research [85]. We expect that the compliance with emerging standards will enhance the portability and interoperability of our system.

Object. An object can model any real-world entity or conceptual entity. Each object has a unique identity which is immutable and persists as long as the object exists. An object has its state and behavior. The state of an object captures

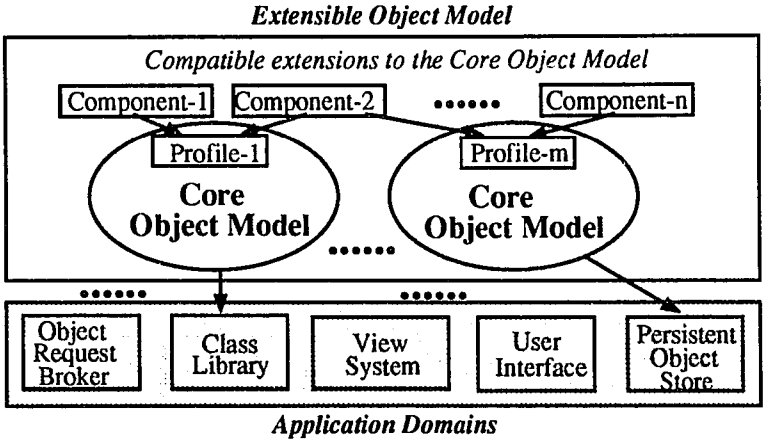

Figure 3.5: The extensible object model (EOM).

the information carried by the object. The behavioral semantics of an object is implemented by operations to change the state of the object. For example, a data object in a database might be used to model the employees of a company. The name of an employee is part of the state of the employee object. There might be an operation defined to compute the average salary for all the employees in the company. Thus an object ρ is defined as a triple: $\rho = \langle oid, \sigma, \Delta \rangle$. In this definition, *oid* is a unique identifier, σ is the state of ρ, and Δ is a set of operations. We have introduced the general notions of an object identifier, the state of an object, and the behavioral semantics of an object. Our intent is to provide definitions at an abstract level that suffice in the context of our discussion. The exact meaning of the state and set of operations will be given by each application domain.

Type. Objects are instances of types. A type has a unique and immutable identity, an interface and a set of instances[2]. The definition of the interface is also

[2]The set of instances of a type is also called the extension of the type.

referred to as the type specification which defines the structures and behavior of the type. A type, τ, is defined as a triple: $\tau = \langle tid, \Omega, \Phi \rangle$. In this definition, *tid* is a unique identifier, Ω is the interface of τ, and Φ is the extension of τ. The equivalence of the types of two objects is determined by the equivalence of the identifiers of the two types.

Class. The implementations of an interface of a type are separated from the specification of the type's interface. The interface of a type may have several different implementations. The combination of the interface of a type and one of its implementations defines a class. A class ς is defined as a triple: $\varsigma = \langle cid, \omega_i, \Phi_i \rangle$. In this definition, *cid* is a unique identifier, ω_i is one of the implementations for the interface of the associated type, and Φ_i is the set of objects created via ς. Suppose Φ is the extension of the type τ which is associated with ς, $\Phi_i \subseteq \Phi$ for all i.

Non-object Types. Instances of non-object types are the atomic values like numbers, characters, strings, and so on. The set of non-object types is different in different application domains and can be specified in profiles. For example, Binary Large OBjects (BLOBs) can be chosen as a non-object type for multimedia applications. The set of all non-objects is denoted as Θ. Suppose the set of all object identifiers is denoted as Π, the set of values that can be manipulated in the core object model is denoted as Υ and $\Upsilon = \Theta \cup \Pi$.

Type Hierarchy and Inheritance. All types, except non-object types, form a type hierarchy. The growth of the type hierarchy is through subtyping. Subtyping creates the relationship between supertypes and subtypes. Inheritance is a

mechanism for code reuse. The specification of subtypes can be inherited from supertypes.

Type System and Object System. We define a type system as a group of types which model a specific application domain. Suppose the set of all types except non-object types is denoted as Ψ, a type system can be denoted as Γ which represents a set of types such that $\Gamma \subseteq \Psi$. An object system is the union of all extensions associated with each type in Γ. An object system can also be represented by a set, $\Lambda = \{\rho \mid \rho = \langle oid_\rho, \sigma, \Delta \rangle, \tau = \langle tid_\tau, \omega, \Phi \rangle, \rho \in \Phi, \tau \in \Gamma\}$

The EOM is extensible in the sense that components can be added and then grouped by profiles to tailor to the requirements of existing and future application domains. Now, we can use the EOM to describe the real-world entities in a multidatabase community. Figure 3.6 lists the definitions of databases, DBMSs, multidatabases and multidatabase systems (MDBs). In Project *Zeus*, we model a DBMS as a database type, $\mathcal{DT}$. A database is modeled as a database object, $\mathcal{D}$, which is an instance of a database type. A multidatabase is a set of database objects that may have different database types. We model a multidatabase as a multidatabase object, $\mathcal{MD}$, which is an instance of a multidatabase type, $\mathcal{MDT}$. A multidatabase system is modeled as a multidatabase type. In Figure 3.7, we show an example multidatabase environment that has seven database objects enclosed by circles, three database types ($\mathbf{DT}_1$, $\mathbf{DT}_2$, and $\mathbf{DT}_3$), two multidatabase objects ($\mathbf{MD}_1$, and $\mathbf{MD}_2$), and one multidatabase type ($\mathbf{MDT}$). The boundaries delineated by solid or dotted lines represent the interfaces and the mappings between adjacent modeling constructs. The func-

Definition 3.1 (Database Object: $\mathcal{D}$)
A database is modeled as a database object, $\mathcal{D}$. $\mathcal{D}$ is denoted as $\langle oid_{\mathcal{D}}, \sigma_{\mathcal{D}}, \delta_{\mathcal{D}} \rangle$. $oid_{\mathcal{D}}$ uniquely identifies $\mathcal{D}$. $\sigma_{\mathcal{D}}$ is the state of $\mathcal{D}$. $\delta_{\mathcal{D}}$ defines the operations that can be applied to $\sigma_{\mathcal{D}}$. $\sigma_{\mathcal{D}}$ is modeled by an object system, $\Lambda_{\mathcal{D}}$, and a type system, $\Gamma_{\mathcal{D}}$; i.e., $\sigma_{\mathcal{D}} = \langle \Lambda_{\mathcal{D}}, \Gamma_{\mathcal{D}} \rangle$. For example, if $\mathcal{D}$ is a relation in a RDBMS, $\Lambda_{\mathcal{D}}$ would be a set of tuples and $\Gamma_{\mathcal{D}}$ would consist of the types of all the attributes defined in the relation. $\diamond$

Definition 3.2 (Database Type: $\mathcal{DT}$)
A database management system (DBMS) is modeled as a database type. A database object is an instance of a database type. A database type, $\mathcal{DT}$, is denoted as $\langle tid_{\mathcal{DT}}, \Omega_{\mathcal{DT}}, \Phi_{\mathcal{DT}} \rangle$. $tid_{\mathcal{DT}}$ uniquely identifies $\mathcal{DT}$. For example, DBMSs like Oracle and ObjectStore, have different $tid_{\mathcal{DT}}$'s. $\Omega_{\mathcal{DT}}$ defines the interface of $\mathcal{DT}$; for example, a query language. $\Phi_{\mathcal{DT}}$ consists of a set of database objects that are instances of $\mathcal{DT}$. $\diamond$

Definition 3.3 (Multidatabase Object: $\mathcal{MD}$)
A multidatabase is modeled as a multidatabase object, $\mathcal{MD}$. $\mathcal{MD}$ is denoted as a triple, $\langle oid_{\mathcal{MD}}, \sigma_{\mathcal{MD}}, \delta_{\mathcal{MD}} \rangle$. $oid_{\mathcal{MD}}$ uniquely identifies $\mathcal{MD}$. $\sigma_{\mathcal{MD}}$ is the state of $\mathcal{D}$. $\delta_{\mathcal{MD}}$ defines the operations that can be applied to $\sigma_{\mathcal{D}}$. $\sigma_{\mathcal{MD}}$ is modeled by an object system, $\Lambda_{\mathcal{MD}}$, and a type system, $\Gamma_{\mathcal{MD}}$. $\Lambda_{\mathcal{MD}} = \{\mathcal{D} \mid \mathcal{D} = \langle oid_{\mathcal{D}}, \sigma_{\mathcal{D}}, \Delta_{\mathcal{D}} \rangle, \mathcal{DT} = \langle tid_{\mathcal{DT}}, \Omega_{\mathcal{DT}}, \Phi_{\mathcal{DT}} \rangle, \mathcal{D} \in \Phi_{\mathcal{DT}}, \Phi_{\mathcal{DT}} \subseteq \Lambda_{\mathcal{MD}}, \mathcal{DT} \in \Gamma_{\mathcal{MD}}, |\Gamma_{\mathcal{MD}}| \geq 1\}$, where $\mathcal{D}$ is a database object, $\mathcal{DT}$ is a database type. For example, a multidatabase object may correspond to a federated database in a federated database system. $\diamond$

Definition 3.4 (Multidatabase Type: $\mathcal{MDT}$)
A multidatabase system is modeled as a multidatabase type. A multidatabase object is an instance of a multidatabase type. A multidatabase type, $\mathcal{MDT}$, is denoted as a triple, $\langle tid_{\mathcal{MDT}}, \Omega_{\mathcal{MDT}}, \Phi_{\mathcal{MDT}} \rangle$. $tid_{\mathcal{MDT}}$ uniquely identifies $\mathcal{MDT}$. For example, multidatabase systems like Pegasus and Multibase have different $tid_{\mathcal{MDT}}$'s. $\Omega_{\mathcal{MDT}}$ defines the interface of $\mathcal{MDT}$; for example, a global query language. $\Phi_{\mathcal{MDT}}$ consists of a set of multidatabase objects which is in turn composed of a set component databases that may have different database types. $\diamond$

Figure 3.6: Definitions of databases, DBMSs, multidatabases, and MDBSs.

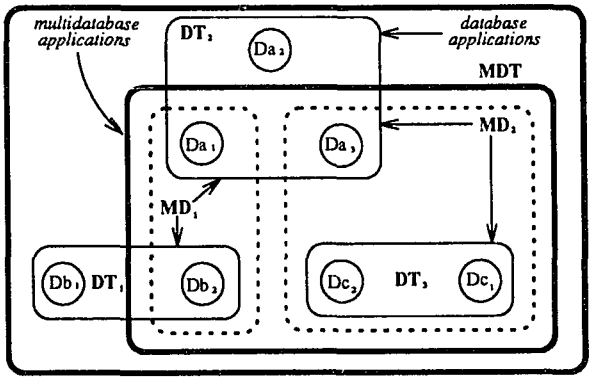

Figure 3.7: An example multidatabase environment.

tionality of a multidatabase system can be deemed as the creation of a linkage or a mapping between a multidatabase type and an extensible set of database types.

3.4.2 Mapping Methodology

Between applications and DBMSs, we need a mapping to ensure that the data is brought from DBMSs to applications in the right format and applications are able to manipulate the data. Our mapping methodology has three major elements; i.e., the structures, the type relations, and the semantics. The mapping of **structures** can translate the modeling constructs from one application domain to another. For example, a relation in a relational database may be mapped to a set of objects in an object database. The mapping of **type relations** is complicated. In Figure 3.8, type-P is the type of a data object stored in a DBMS and type-R is the type of a data object used by an application. We refer to the DBMS as the provider and the application as the receiver. We identify three kinds of type relations as examples and we use Figure 3.8 to explain type relations in a Provider-Receiver model. First of all, type-P and type-R may be *equivalent.* In this case, state-P can be moved to

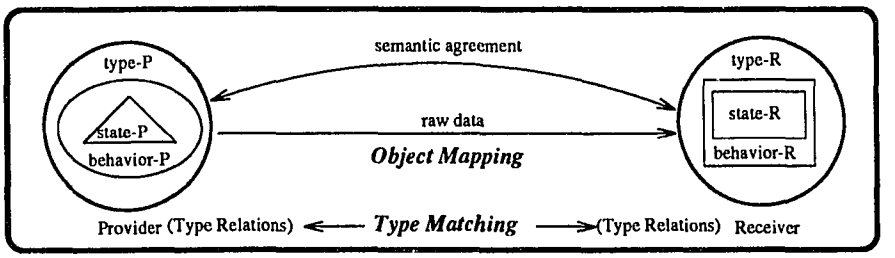

Figure 3.8: The Provider-Receiver Model and the Mapping Methodology.

the receiver site and becomes **state-R**. **Behavior-R** is directly applicable to **state-R**. Second, **type-P** and **type-R** may *match* in their behavioral semantics. However, either **state-P** needs to be transformed or **behavior-P** needs to be translated to ensure an exact match. Third, there might be no types in the receiver site that match or are equivalent to **type-P** in the provider site. In this case, the receiver may define a new type that matches or is equivalent to **type-P**. Or the receiver may instantiate a proxy object that is able to invoke remote operations on the provider site. In any of these examples, there must be a semantic agreement between the provider and the receiver. The mapping of **semantics** is partly captured by type relations. A global request along with the resulting type matchings and object mappings determine the mapping of semantics.

The abstraction constructs in the layered architecture should convey enough information for mapping the structure, behavior, and semantics of data from DBMSs to applications. The $\mathcal{ZVS}$ must be able to determine whether an abstraction construct is capable of producing semantically meaningful data for applications. Figure 3.8 shows how the type relations, type matchings, and object mappings can be viewed as our mapping methodology and fit in the Provider-Receiver Model. Now we can explain how the mapping methodology is embedded in the *Zeus* view constructs.

```
define LocalInterface MileageAward as {
    structure { type MileageAwardMember:tuple
                        (name:string,
                         address:tuple(street:string,city:string),
                         card#:integer,
                         flights:set(Flight))
                type Flight:tuple
                        ( flightNo:string, date:string )
    }
    access    {type MileageAwardMember <=>
                    type MileageAwardMember in ODBMS,
               typeFlight <=>  type Flight in ODBMS
    }
    semantic  {flightNo of type Flight <=>
                    flight# of type FLIGHT-INFO in Reservation
    }
} // End of MileageAward definition
```

```
define LocalInterface Reservation as {
    structure { type Customer:set(RESERVATION)
                type RESERVATION:tuple(name:string,
                    flight#:string,date:string,confirmation#:integer)
                type Flight-Info:set(FLIGHT-INFO)
                type FLIGHT-INFO:tuple(flight:string,from:string,
                                       to:string,mileage:integer)
    }
    access    type RESERVATION <=>
                    type reservation in RDBMS,
              type FLIGHT-INFO <=>
                    type flight-info in RDBMS
    }
    semantic  {flight# of type FLIGHT-INFO <=>
               flightNo of type FLIGHT in MileageAwardMember
    }
} // End of Reservation definition
```

Figure 3.9: Example 3.1.

Example 3.1. The local interfaces for the databases of our running example are listed in Figure 3.8. Each database has a corresponding local interface. Each local interface has three major components; i.e., the structure, the access mechanism, and the semantics. Notice that although the database definitions appeared in Figure 3.4 have different syntactic structures for different DBMSs, the corresponding local interfaces are all defined using a uniform syntax. In Figure 3.9, we use "<=>" to denote the orderly correspondence between the attributes of two types. The details of such a correspondence are defined in terms of type relations and object mappings.

Figure 3.10 gives formal definitions for a view type and a local interface. All the *Zeus* view constructs are instances of the *Zeus* view type, $\mathcal{ZVT}$. We use $\mathcal{ZVT}$ to group commonalities of the *Zeus* view constructs and specify a common management interface for the *Zeus* views. A local interface describes a local schema of a participating database in the context of a global data model. A request $\mathcal{R}$ against a

local interface $\mathcal{LI}$ is translated to a set of local database access routines. A transaction created by the multidatabase system coordinates the execution of these database access routines and makes sure that the result matches the semantics of the request. Since a request against a local interface might be originally generated by a request against a base view or a view, the transaction is created upon the arrivial of the original request. Local interfaces hide the heterogeneity and distribution of participating DBMSs from the upper layers of the layered architectures. The translation of a request against a local interface is handled by the $\mathcal{ZMS}$ server which will be described in Section 4.2.4.

Example 3.2. In Figure 3.11, we define a base view, $\mathcal{BV}_1$, for application#2. $\mathcal{BV}_1$ extracts the flight information from the relational database when the customers use their MileageAward card to check in for their flights. Application#2 uses $\mathcal{BV}_1$ to automate the update of the flight record in the object database. In Figure 3.11, we use "<= input" to denote that the value of card# will be fed by the application. Since flight# and date are tied to the local interface, MileageAward, the access mechanism of MileageAward can be used to update the flight record.

Base views are derived from local interfaces to describe the data that the local DBMS is willing to import or export. A request $\mathcal{R}$ against a base view $\mathcal{BV}$ is translated to a request $\mathcal{R}'$ against a local interface $\mathcal{LI}$. On the other hand, an application can use base views to control how the data should be presented. For example, an application may use different base views to export data from the same database. This is a way to achieve abstraction relativism. Figure 3.12 gives a formal defini-

Definition 3.5 (View Type: $\mathcal{VT}$)
A view type, $\mathcal{VT}$, is denoted as $\langle tid_{\mathcal{VT}}, \Omega_{\mathcal{VT}}, \Phi_{\mathcal{VT}} \rangle$. $\Omega_{\mathcal{VT}}$ defines the interface of $\mathcal{VT}$. For example, the operations for managing the view objects. $\Phi_{\mathcal{VT}}$ consists of all the view objects which are instances of $\mathcal{VT}$. $\diamond$

Definition 3.6 (Local Interface: $\mathcal{LI}$)
A local interface, $\mathcal{LI}$, is an instance of the *Zeus* view type, $\mathcal{ZVT}$. $\mathcal{LI}$ is a view object denoted as $\langle oid_{\mathcal{LI}}, \sigma_{\mathcal{LI}}, \delta_{\mathcal{LI}} \rangle$. $oid_{\mathcal{LI}}$ uniquely identifies $\mathcal{LI}$. $\sigma_{\mathcal{LI}}$ is modeled as a tuple of four elements; i.e., $\sigma_{\mathcal{LI}} = \langle \mathcal{LI}_s, \mathcal{LI}_a, \mathcal{LI}_t, \mathcal{LI}_m \rangle$, where $\mathcal{LI}_s$ denotes the structure of $\sigma_{\mathcal{LI}}$, $\mathcal{LI}_a$ denotes the access mechanism of $\sigma_{\mathcal{LI}}$, $\mathcal{LI}_t$ denotes the type relations of $\sigma_{\mathcal{LI}}$, and $\mathcal{LI}_m$ denotes the semantics of $\sigma_{\mathcal{LI}}$. $\mathcal{LI}_s = \mathcal{S}(\mathcal{LI})$, where $\mathcal{S}(\mathcal{LI})$ denotes the type definitions that constitute the structure of $\sigma_{\mathcal{LI}}$. Let $\oplus$ be an operator that composes a set of database access functions in a certain order.

$$\mathcal{LI}_a : \varphi \mapsto \oplus_i \delta_i, \text{ where } \delta_i \in \{\kappa_i \mid \kappa_i(\mathcal{D}_i) \subseteq \Lambda_{\mathcal{D}_i}\}$$

φ provides a syntactic description of the data to be retrieved from local DBMSs. κ_i is a local database access function that is applied to a database object, i.e., $\mathcal{D}_i$, and returns a subset of the data in database $\mathcal{D}_i$. $\mathcal{LI}_a$ defines a mapping that maps φ to a composition of local database access functions.

$$\mathcal{LI}_t : \{\chi_i \mid \chi_i = (oid_{\mathcal{D}_i} :: (\mathcal{S}(\mathcal{LI}_i) = \Gamma_{\mathcal{D}_i}))\}$$

In other words, $\mathcal{LI}_t$ is represented by a set of type translations. Each type translation, χ_i, defines a set of type relations between the types in a database object $\mathcal{D}_i$, denoted as $\Gamma_{\mathcal{D}_i}$, and the types in a local interface $\mathcal{LI}_i$, denoted as $\mathcal{S}(\mathcal{LI}_i)$. $\mathcal{S}(\mathcal{LI}_i) = \Gamma_{\mathcal{D}_i}$ denotes a set of type relations between two sets of types; i.e., $\mathcal{S}(\mathcal{LI}_i)$ and $\Gamma_{\mathcal{D}_i}$. The notation, ::, denotes the binding between the set of type relations and the object identifier $oid_{\mathcal{D}_i}$ of a database object $\mathcal{D}_i$. $\delta_{\mathcal{LI}}$ specifies the object mapping and other runtime behaviors. The semantics of $\mathcal{LI}$, $\mathcal{LI}_m$, is embedded in the definitions of both $\sigma_{\mathcal{LI}}$ and $\delta_{\mathcal{LI}}$. $\diamond$

Figure 3.10: Definitions of View Type and Local Interface.

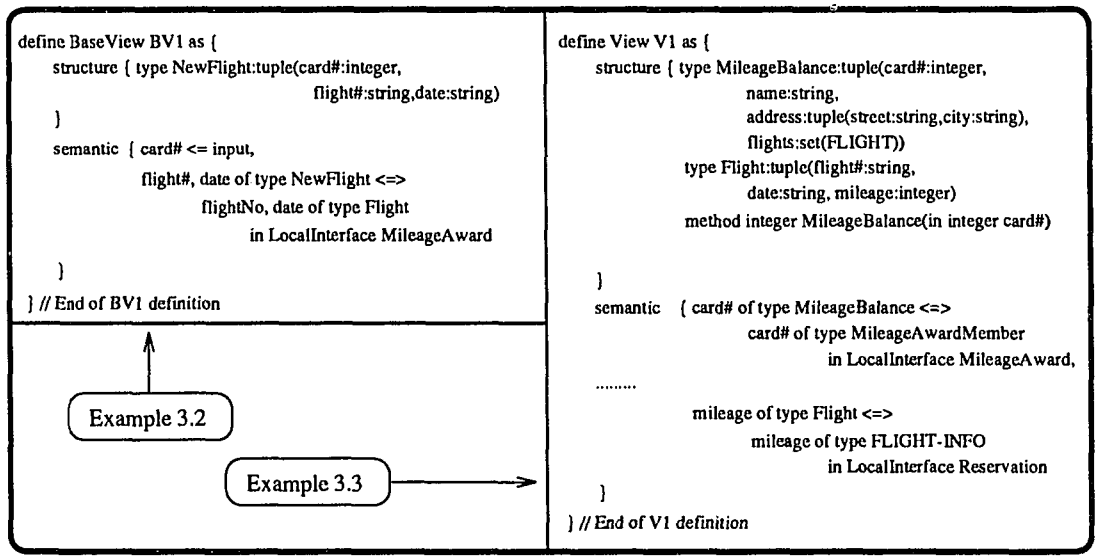

Figure 3.11: Examples 3.2 and 3.3.

tion for a base view. The translation of a request against a base view to a request against a local interface is also handled by the $\mathcal{ZMS}$ server. For those applications that require integrated access to multiple databases, local interfaces and base views are not enough for modeling their needs. We introduce another layer of abstraction; i.e., views, to model integrated access through the integration of existing $\mathcal{Zeus}$ view constructs. We describe views in an integration model.

3.4.3 Integration Model

The role of an integration model is to express the conceptual relationships between the integration mechanism and the integration semantics. Our approach filters out the heterogeneity of underlying DBMSs through the mapping methodology that preserves the semantics and interrelationships of participating DBMSs. After the mapping, we have a single basic structure, the view object. Having a single basic

Definition 3.7 (Base View: $\mathcal{BV}$)

A base view, $\mathcal{BV}$, is an instance of the *Zeus* view type, $\mathcal{ZVT}$. $\mathcal{BV}$ is a view object denoted as $\langle oid_{\mathcal{BV}}, \sigma_{\mathcal{BV}}, \delta_{\mathcal{BV}} \rangle$. $oid_{\mathcal{BV}}$ uniquely identifies $\mathcal{BV}$. $\sigma_{\mathcal{BV}}$ is modeled as a triple; i.e., $\sigma_{\mathcal{BV}} = \langle \mathcal{BV}_s, \mathcal{BV}_t, \mathcal{BV}_m \rangle$, where $\mathcal{BV}_s$ denotes the structure of $\sigma_{\mathcal{BV}}$. $\mathcal{BV}_t$ denotes the type relations of $\sigma_{\mathcal{BV}}$, and $\mathcal{BV}_m$ denotes the semantics of $\sigma_{\mathcal{BV}}$.

$$\mathcal{BV}_s = \mathcal{S}(\mathcal{BV}), \text{ where}$$

$\mathcal{S}(\mathcal{BV})$ denotes the type definitions that constitute the structure of $\sigma_{\mathcal{BV}}$.

$$\mathcal{BV}_t : \{\chi_i \mid \chi_i = (oid_{\mathcal{BV}} :: (\mathcal{S}(\mathcal{BV}) = \mathcal{S}(\mathcal{LI}_i)))\}$$

In other words, $\mathcal{BV}_t$ is represented by a set of type translations. Each type translation, χ_i, defines a set of type relations between the types in a base view $\mathcal{BV}$, denoted as $\mathcal{S}(\mathcal{BV})$, and the types in a local interface $\mathcal{LI}_i$, denoted as $\mathcal{S}(\mathcal{LI}_i)$. $\mathcal{S}(\mathcal{BV}) = \mathcal{S}(\mathcal{LI}_i)$ denotes a set of type relations between two sets of types; i.e., $\mathcal{S}(\mathcal{BV})$ and $\mathcal{S}(\mathcal{LI}_i)$. The set of type relations is bound to the object identifier $oid_{\mathcal{BV}}$ of a base view $\mathcal{BV}$. $\delta_{\mathcal{BV}}$ specifies the object mapping and other runtime behaviors. The semantics of $\mathcal{BV}$, $\mathcal{BV}_m$, is embedded in the definitions of both $\sigma_{\mathcal{BV}}$ and $\delta_{\mathcal{BV}}$. $\diamond$

Figure 3.12: Definition of Base View.

structure facilitates the integration process. In the remainder of this section, we explain what the integration model is and how it fits in the $\mathcal{ZVM}$.

Example 3.3. We define a view, $\mathcal{V}_1$, for application#1. $\mathcal{V}_1$ retrieves the flight record from the object database. The mileage of each flight is retrieved from the relational database. Since application#1 can use $\mathcal{V}_1$ to access the mileage for each flight in the flight record, the total mileage balance can be computed. The method, MileageBalance, takes the card# as the input argument and returns the mileage balance.

We provide a formal definition of a view in Figure 3.12. Views are derived from existing views, base views, and local interfaces. This derivation is defined by an integration model. A request $\mathcal{R}$ against a view $\mathcal{V}$ is mapped to a number of requests. Each of these requests might be against other view, a base view or a local interface. The mapping is determined by how the view is constructed. For example, a view might be composed of a base view for exporting data from a voucher database and another base view for importing data to a purchase order database. We may issue a request against this view to close the purchase orders whose corresponding vouchers have come through. This request will spawn a request against the base view that exports new vouchers and another request against the base view that imports purchase order numbers for closing purchase orders. The entire process is coordinated by a transaction. The translation of a request against a view to other requests is handled by the $\mathcal{ZMS}$ server. Now we can describe the integration model which provides the basis for deriving views from existing views, base views, and local

Definition 3.8 (View: $\mathcal{V}$)

A view, $\mathcal{V}$, is an instance of the *Zeus* view type, $\mathcal{ZVT}$. $\mathcal{V}$ is a view object denoted as $\langle oid_{\mathcal{V}}, \sigma_{\mathcal{V}}, \delta_{\mathcal{V}} \rangle$. $oid_{\mathcal{V}}$ uniquely identifies $\mathcal{V}$. $\sigma_{\mathcal{V}}$ is modeled as a tuple of four elements; i.e., $\sigma_{\mathcal{V}} = \langle \mathcal{V}_s, \mathcal{V}_i, \mathcal{V}_t, \mathcal{V}_m \rangle$, where $\mathcal{V}_s$ denotes the structure of $\sigma_{\mathcal{V}}$. $\mathcal{V}_i$ denotes the integration mechanism of $\sigma_{\mathcal{V}}$. $\mathcal{V}_t$ denotes the type relations of $\sigma_{\mathcal{V}}$, and $\mathcal{V}_m$ denotes the semantics of $\sigma_{\mathcal{V}}$.

$$\mathcal{V}_s = \mathcal{S}(\mathcal{V}), \text{ where}$$

$\mathcal{S}(\mathcal{V})$ denotes the type definitions that constitute the structure of $\sigma_{\mathcal{V}}$. Let $\mathcal{S}(\nu_i)$ be the set of types that constitue the structure of a view object ν_i, and ν_i belongs to the set of all view objects of the *Zeus* view type; i.e., $\Lambda_{\mathcal{ZVT}}$. Let $\mathcal{Q}$ denote the set of all the integration operators and type constructors supported by the integration model. Let $\otimes$ be an operator, M be a set of types, and W be a set of type constructors and integration operators. $M \otimes W$ denotes all possible ways of applying the constructors or operators in W to the types in M. Now, we can define $\mathcal{V}_i$ as follows:

$$\mathcal{V}_i : syntactic - descriptions \mapsto ((\bigcup_i \mathcal{S}(\nu_i)) \bigcup \Lambda_{\mathcal{ZVT}}) \otimes \mathcal{Q}$$

In other words, $syntactic - descriptions$ describes the integration mechanism. $\mathcal{V}_i$ maps $syntactic - descriptions$ to the actual integration mechanism. The type relations, $\mathcal{V}_t$, can be denoted as a set:

$$\mathcal{V}_t : \{\chi_i \mid \chi_i = oid_{\mathcal{V}} :: \mathcal{S}(\mathcal{V}) = \mathcal{S}(\nu_i), \nu_i \in \Lambda_{\mathcal{ZVT}}\}$$

In other words, $\mathcal{V}_t$ is represented by a set of type translations. Each type translation, χ_i, defines a set of type relations between the types in a view $\mathcal{V}$, denoted as $\mathcal{S}(\mathcal{V})$, and the types in a view object ν_i, denoted as $\mathcal{S}(\nu_i)$. $\Lambda_{\mathcal{ZVT}}$ denotes the set of all view objects. $\mathcal{S}(\mathcal{V}) = \mathcal{S}(\nu_i)$ denotes a set of type relations between two sets of types; i.e., $\mathcal{S}(\mathcal{V})$ and $\mathcal{S}(\nu_i)$. The set of type relations is bound to the object identifier $oid_{\mathcal{V}}$ of a view $\mathcal{V}$. $\delta_{\mathcal{V}}$ specifies the object mapping and other runtime behaviors. The integration semantics of $\mathcal{V}$, $\mathcal{V}_m$, is embedded in the definitions of both $\sigma_{\mathcal{V}}$ and $\delta_{\mathcal{V}}$. $\diamond$

Figure 3.13: Definition of View.

interfaces. An integration model is used to deal with the integration issues that can be classified along two dimensions; i.e., data integration and control integration:

1. **Data Integration.** There are a number of issues involved when the data from multiple databases are integrated [41]; for example, homonym, synonym, scale difference, type difference, missing data, conflicting values, semantic difference, structural difference, etc. The mappings between different object types may need to be defined to achieve the integration through object transformations. The multidatabase system must guarantee that the integrated data is semantically meaningful to applications.

2. **Control Integration.** The behavior semantics of data objects and application objects should be preserved or translated during the integration. The multidatabase system must also keep track of the configuration of participating DBMSs in order to coordinate the cooperation in different tasks; for example, concurrency control, transaction management, security, recovery, etc. Control integration must ensure that the multidatabase system maintains consistent runtime behavior.

Our integration model is a compatible extension of the EOM. There are two aspects in our integration model; i.e., the structural aspect and the semantic aspect. The integration model must have the expressiveness to represent various structural integrations; for example, tuple constructor (aggregation), set constructor (grouping), union of types (generalization), etc. These are examples of type constructors that integrate the types defined in view objects. Similarly, view objects may be directly integrated through integration operators. For example, a view can be defined as the

composition of a base view for exporting the flight information from the relational database, and another base view for importing the flight information into the object database. The integration issues are resolved in the definition of *Zeus* views. The semantic aspect of the integration model ensures that the integration resulted from the type constructors and integration operators preserves the integration semantics. We have briefly described an abstract integration model. There are a number of issues that are better addressed by a concrete model. For example, how does the $\mathcal{ZVS}$ check whether a view definition is robust in terms of its relationships with other views and its runtime semantics? How does the $\mathcal{ZVS}$ maintain the consistency of views in the view repositories? These issues are currently being investigated in Project *Zeus*.

3.4.4 Computation Model (I)

The computation model of the $\mathcal{ZVS}$ describes how the $\mathcal{ZVS}$ interprets requests and coordinates the handling of data. In this section, we describe the computation model of the $\mathcal{ZVS}$ in a more abstract fashion. We will provide a concrete description of the computation model in Chapter 4. We start with the OR model which is part of an object-relational interface developed at Iowa State University [90]. The OR model provides a formal basis for describing the integration abstraction, the integration mechanism, and the database access mechanism on top of a RDBMS and an ODBMS. We describe the OR model as an extension to the EOM. There are three basic constructs in the OR model; i.e., a template, a portal, and a view. Formal definitions of these constructs are listed in Figure 3.14. A template models a generic interface of a database type. Each database type may have multiple interfaces. The basic idea is to automate the generation of DBMS access routines by substituting the

Definition 3.9 (Template: $\mathcal{T}$)
A template, $\mathcal{T}$, is modeled as a template object. $\mathcal{T}$ is denoted as a triple: $\langle oid_{\mathcal{T}}, \sigma_{\mathcal{T}}, \Delta_{\mathcal{T}} \rangle$, where $oid_{\mathcal{T}}$ uniquely identifies $\mathcal{T}$, $\sigma_{\mathcal{T}}$ is state of $\mathcal{T}$, and $\Delta_{\mathcal{T}}$ is the behavior of $\mathcal{T}$. $\sigma_{\mathcal{T}}$ has three major components; i.e., the substitution specifications ($\sigma_{\mathcal{T}_s}$), the related database type ($\sigma_{\mathcal{T}_{\mathcal{DT}}}$), and the associated template file ($\sigma_{\mathcal{T}_f}$). $\Delta_{\mathcal{T}}$ defines the set of the methods that can be applied to $\sigma_{\mathcal{T}}$. $\diamond$

Definition 3.10 (Portal: $\mathcal{P}$)
A portal object, $\mathcal{P}$, is modeled as a triple: $\langle oid_{\mathcal{P}}, \sigma_{\mathcal{P}}, \Delta_{\mathcal{P}} \rangle$, where $oid_{\mathcal{P}}$ uniquely identifies $\mathcal{P}$, $\sigma_{\mathcal{P}}$ is the state of $\mathcal{P}$, and $\Delta_{\mathcal{P}}$ is the behavior of $\mathcal{P}$. $\sigma_{\mathcal{P}}$ defines a pair of program objects, denoted as $\langle \mathcal{PO}_1, \mathcal{PO}_2 \rangle$, and their related database types. Semantically, $\mathcal{P}$ describes how to move and transform the data from one DBMS to another in a way consistent with application semantics. Functionally, $\mathcal{PO}_i$, where $i = 1$ or 2, describes a mapping, $\mathcal{M}_{\mathcal{PO}_i} : \kappa_i(\mathcal{A}) \mapsto \mathcal{B}$, where $\mathcal{A}$ an $\mathcal{B}$ are either a set of objects or a flat file. κ_i is a function that takes $\mathcal{A}$ as its parameter, and $\mathcal{B}$ is the result of the mapping. Possible combinations of $\mathcal{A}$ and $\mathcal{B}$ are $\langle \mathcal{A}, \mathcal{B} \rangle = \langle \Lambda_{\mathcal{D}_i}, \mathcal{F} \rangle$ or $\langle \mathcal{F}, \Lambda_{\mathcal{D}_i} \rangle$, where $\Lambda_{\mathcal{D}_i}$ is the set of objects stored in database object, $\mathcal{D}_i$, and $\mathcal{F}$ is a flat file. $\diamond$

Definition 3.11 (OR View: $\mathcal{V}_{OR}$)
An OR view, $\mathcal{V}_{OR}$, is denoted as a triple: $\langle oid_{\mathcal{V}_{OR}}, \sigma_{\mathcal{V}_{OR}}, \Delta_{\mathcal{V}_{OR}} \rangle$, where $oid_{\mathcal{V}_{OR}}$ uniquely identifies $\mathcal{V}_{OR}$, $\sigma_{\mathcal{V}_{OR}}$ is the state of $\mathcal{V}_{OR}$, and $\Delta_{\mathcal{V}_{OR}}$ is the behavior of $\mathcal{V}_{OR}$. $\sigma_{\mathcal{V}_{OR}}$ describes the structures defined in $\mathcal{V}_{OR}$, denoted as $\mathcal{S}(\mathcal{V}_{OR})$, the type relations between $\mathcal{S}(\mathcal{V}_{OR})$ and $\Gamma_{\mathcal{D}_i}$, and the selection criteria for the data to be retrieved from the database object $\mathcal{D}_i$. $\Gamma_{\mathcal{D}_i}$ denotes the types of the objects stored in database object, $\mathcal{D}_i$. $\diamond$

Figure 3.14: Definitions of Template, Portal and OR View.

constructs in a template file based on a user-defined view. The resulting template file tailors to the requirements of a specific application and becomes a customized interface to the DBMS. Stonebraker [102] introduced the notion of a portal in POSTGRES as an intermediate construct between applications and a DBMS. The function of a portal is to provide a user access to the set of objects generated on a controlled basis. In the OR Interface, a portal is derived from an OR view and a template, and serves as a runtime construct that may access a RDBMS or an ODBMS. A portal, $\mathcal{P}$, is modeled as a portal object. In Definition 3.10, $\mathcal{M}_{\mathcal{PO}_i}$ is derived from the interface of the DBMS and the methods defined in the objects of

related databases. In the OR Interface, views are used to facilitate the integration of data of different types. An object based view is a virtual relation where the data is stored in an object type within an object database. A relation based view is a virtual object where the data is stored in one or more relations in a relational database. In the OR model, an OR view is modeled as a view object. The OR Interface system guarantees that given an OR view and a template, there is a consistent and well-defined mapping that maps the view definition and the template file to a pair of DBMS access routines; i.e., a portal. The notion of an OR view is different from the view defined in Definition 3.8 in the sense that an OR view only models the integration between RDBMSs and ODBMSs. The notion of a view defined in Definition 3.8 models the integration of any types of DBMSs.

3.4.4.1 Generalization of the OR Model We can generalize the OR model to provide another level of detail in the theoretical foundation of the $\mathcal{ZVM}$; i.e., exactly how local interfaces generate local database access routines. Notice that the *Zeus* view is similar to the OR view in the sense that both constructs describe integrated access to heterogeneous DBMSs. However, the *Zeus* view is more generic and supports multiple levels of abstraction. Therefore, we replace the OR view by the *Zeus* view. We then tie the OR model to the integration model by linking the notions of template and portal to the local interface. In Figure 3.15, we provide definitions for a DBMS interface and a multidatabase interface. As can be seen from Definition 3.12, the notion of OR views is replaced by the notion of a global request against a local interface. The notion of portals exists as the supporting runtime constructs behind the mapping, $\mathcal{M}_{\mathcal{T}_i}$. In Definition 3.13, the interface of a multidatabase type

Definition 3.12 (DBMS Interface: $\mathcal{DI}$)
A DBMS interface, $\mathcal{DI}$, is denoted as a set of pairs. Each pair is denoted as $\langle \mathcal{T}_i, \mathcal{M}_{\mathcal{T}_i} \rangle$, where $\mathcal{T}_i$ is a template, and $\mathcal{M}_{\mathcal{T}_i}$ is a map satisfying:

$$\mathcal{M}_{\mathcal{T}_i} : \langle \mathcal{T}_i, \mathcal{R}, \mathcal{LI} \rangle \mapsto \Lambda'_{\mathcal{D}_i}, \text{ where}$$

$\mathcal{R}$ is a request against a local interface ($\mathcal{LI}$) associated with $\mathcal{DI}$, $\Lambda_{\mathcal{D}_i}$ is the set of objects stored in a database object ($\mathcal{D}_i$), and $\Lambda'_{\mathcal{D}_i} \subseteq \Lambda_{\mathcal{D}_i}$. $\diamond$

Definition 3.13 (Multidatabase Interface: $\mathcal{MI}$)
Given a global request, $\mathcal{R}$, and a set of view objects referenced in $\mathcal{R}$; i.e., $\mathcal{VO}$. $\mathcal{MI}$ derives a sequence of well-defined mappings, $\Pi_{\mathcal{MI}}$, denoted as:

$$\Pi_{\mathcal{MI}} : \langle \mathcal{R}, \mathcal{VO} \rangle \mapsto \{\alpha_i | \alpha_i = \langle \mathcal{T}_i, \mathcal{R}_i, \mathcal{LI}_i \rangle\} \mapsto \bigcup \Lambda'_{\mathcal{D}_i}, \text{ where}$$

$\mathcal{R}_i$ is a request, $\mathcal{LI}_i$ is a local interface, and $\bigcup \Lambda'_{\mathcal{D}_i}$ is a subset of the objects belonging to the accessed database objects. $\diamond$

Figure 3.15: Definitions of DBMS Interface and Multidatabase Interface.

is defined as a sequence of mappings. Based on the above definitions, we can view the computation model of the $\mathcal{ZVS}$ as a functional composition of $\mathcal{MI}$ and a set of $\mathcal{DI}$ plus a request as a parameter:

$$\mathcal{CM}(\mathcal{R}) = [\mathcal{MI} \circ \partial(\bigcup_i \mathcal{DI}_i)](\mathcal{R}), \text{ where}$$

$\mathcal{CM}$ denotes the computation model as a function that takes a request $\mathcal{R}$ as the parameter. The notation, $\circ$, represents the functional composition of mappings. The notation, ∂, denotes a selected subset of the set following ∂; i.e., $\partial(\bigcup_i \mathcal{DI}_i) \subseteq \bigcup_i \mathcal{DI}_i$. Figure 3.16 gives an example that illustrates the interfaces and mappings we just defined for the computation model of the $\mathcal{ZVS}$. The interfaces and mappings of database objects, multidatabase objects, and database types have concrete object and runtime semantics. On the contrary, the interfaces and mappings of a multidatabase type only has abstract object and behavioral semantics. There must exist a well-

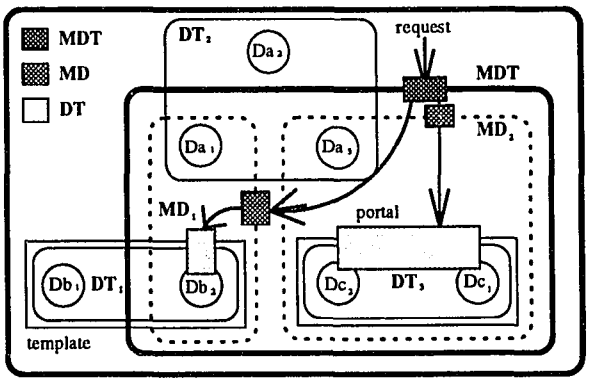

Figure 3.16: The mappings in the $\mathcal{ZVS}$.

defined mapping such that a global request can be mapped to runtime constructs. This is defined and verified within the theoretical foundation. The shaded area in Figure 3.16 illustrates how a global request is turned into runtime constructs after reaching the interfaces of $\mathcal{MDT}$, $\mathcal{MD}_1$, $\mathcal{MD}_2$, $\mathcal{DT}_1$ and $\mathcal{DT}_3$.

3.5 Summary

The notion of *Zeus* views has abstract object semantics. The *Zeus* views are independent of the representations and storage of the data involved. The representation of *Zeus* views is stable; i.e., independent of the size or type of the described data. The *Zeus* views are uniform since they are all modeled by a common data model, i.e., the EOM. The *Zeus* views describe the mappings between DBMSs and applications in terms of type translations and object mappings which are adjustable to application semantics and independent of participating DBMSs. Base views and views are freely composable to achieve different degrees of integration and different levels of interoperation among participating DBMSs. This allows us to support semantic relativism in the $\mathcal{ZVM}$. The *Zeus* views are sharable across organizational

and administrative domains. The effect of computing environments is separated from the definitions of the *Zeus* views. This is how we provide the basis for distributed abstraction modeling. All the above characteristics of the $\mathcal{ZVM}$ explain how the $\mathcal{ZVM}$ provides a solution to the modeling problem that we discussed in Chapter 1. Both the theoretical foundation and the computation model have concrete semantics implemented by the $\mathcal{ZVS}$. In Chapter 4, we will elaborate on how we materialize these concepts and solve the design problem that we addressed in Chapter 1.

CHAPTER 4. THE ZEUS VIEW SYSTEM

The short-term goal of Project *Zeus* is to develop a multidatabase system, the *Zeus Multidatabase System* (*ZMS*), as a test bed for the *ZVM*. The complexity of developing such a large-scale distributed system calls for a complete method that provides a solution for the system design problem. We chose to apply object-oriented techniques and software engineering concepts in our method. To apply object-oriented techniques, we introduce CORBA [122] to provide a distributed object infrastructure. The concrete part of our computation model relies on such an infrastructure to attach runtime semantics to the *Zeus* views. To apply software engineering concepts, we introduce the notion of frameworks to promote design and code reuse. Our method allows us to achieve an extensible and portable architectural design for the *ZMS*. For the remainder of this Chapter, we will elaborate on our method and the architectural design of the *ZMS*. A number of implementation issues will be discussed based on our established experience with the development of the *ZMS*.

4.1 Methodology

Our method falls into the category of methodologies in object-oriented design and analysis [42]. We start with the description of a distributed object infrastructure

based on CORBA [122]. We show how a concrete computation model is created by integrating this infrastructure with the $\mathcal{ZVM}$.

Although conventional procedural and structured programming has provided an approach to modularized design, it is not powerful enough for dealing with the complexity of large-scale programming. To deal with the complexity of the $\mathcal{ZMS}$ development, we decided to use frameworks. Ralph Johnson gave a definition of frameworks in [116]: "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems." The use of frameworks adds five attributes to our approach that make the development of large-scale distributed systems feasible; i.e., code/design reuse, portability, extensibility, and maintainability. Code/design reuse is the result of grouping commonalities in frameworks. High-level frameworks contain reusable abstract design which leads to low-level code reuse through class libraries. Portability is achieved through the separation of machine-dependent and machine-independent design. The machine-independent design is fully portable. Extensibility refers to the ability to extend and specialize functionality such that the structure and behavioral semantics of the system design can be selectively changed. Since running $\mathcal{ZMS}$s share high-level design specifications, the interoperability among abstract declarative interfaces carries through low-level class libraries which in turn lays out a basis for interoperation among $\mathcal{ZMS}$s. The above-mentioned attributes simplify the development of the $\mathcal{ZMS}$ and increase the maintainability of the $\mathcal{ZMS}$ due to the close correlation between the abstract design and the actual system code.

The runtime $\mathcal{ZMS}$ can be described by a client-server model. There is a direct correspondence bewteen the classes in frameworks and the components of the $\mathcal{ZMS}$.

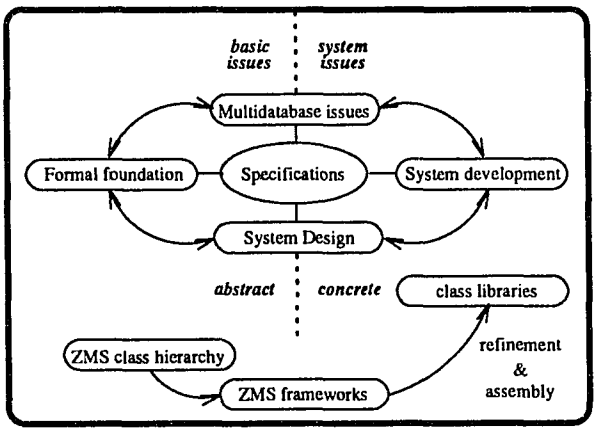

Figure 4.1: Our method in the development of the $\mathcal{ZMS}$.

We will discuss common design dimensions for network servers and explain the client-server architecture of the $\mathcal{ZMS}$ later in this Chapter. Figure 4.1 gives an overview of the $\mathcal{ZMS}$ development and shows where our method fits in the overall picture. The development of the $\mathcal{ZMS}$ starts with a formal foundation; i.e., the $\mathcal{ZVM}$, that deals with basic multidatabase issues. The solution provided by the formal foundation is then embedded in the system design. Our system design is object-oriented. The concrete system design is achieved through the refinement and assembly of frameworks that provides the class libraries for system development. As soon as a runtime system has been assembled, the result of the system test will be useful for adjusting or expanding the formal foundation or the system design. The entire development cycle can be documented through a system specification.

4.1.1 Distributed Object Infrastructure

The object concept has been applied to the foundation and design of the $\mathcal{ZMS}$. In [81], the object technology is identified as an effective approach for controlling

the complexity of heterogeneous distributed computing systems. The $\mathcal{ZMS}$ is a large-scale distributed system. We have selected the Common Object Request Broker Architecture (CORBA) [122] as the basis of our distributed object infrastructure in order to apply the object concept to our approach. In this section, we describe how a CORBA-compliant object request broker (ORB) can achieve wide-area distributed abstraction defined in Chapter 3 and provide the runtime support for the $\mathcal{ZMS}$. We will tie this runtime support to the computation model of the $\mathcal{ZMS}$. First of all, we want to briefly describe what CORBA is. The Common Object Request Broker Architecture, produced by the Object Management Group, is a standard that specifies an architecture for distributed object management. The ORB provides the mechanisms by which objects transparently make requests and receive responses. As stated in [129], the ORB provides interoperability between applications on different machines in heterogeneous distributed environment and seamlessly interconnects multiple object systems. A review and analysis of CORBA can be found in [81]. CORBA provides a declarative specification language, the interface definition language (IDL). Specifications of CORBA objects are separated from their implementations. Many commercial implementations of ORB have become available in recent years. We will explain how a CORBA-compliant implementation of ORBs fits in the $\mathcal{ZMS}$ in Section 4.1.4.

Support for Distributed Abstraction. The support for distributed abstraction via the object concept has a number of aspects. First of all, our notion of views has abstract object semantics. The construction of views from existing views provides multiple levels of abstraction. CORBA objects supports abstrac-

tion. Second, we combine the object-oriented concepts with the layering technique. The application semantics, services, and the $\mathcal{ZMS}$ components are encapsulated in well-defined interfaces. The interfaces are further organized in layers or modules. CORBA objects supports encapsulation through interface specifications. Third, our view mechanism supports different kinds of transparency. Applications are shielded from the heterogeneity, distribution, and autonomy of distributed components. Each distributed component is modeled as an object whose properties are encapsulated by a well-defined interface. The interactions among the distributed components only depend on the interfaces, not on their locations or internals. This implements transparency to heterogeneity and distribution. Each distributed component may change its internal structure without changing its interface. This implements the autonomy of distributed components. Therefore, transparency can also be achieved by CORBA objects. We use the object-level save and restore to facilitate the management of object repositories. The source that deals with the I/O is also reduced in size. Object persistence can be built on top of CORBA. Finally, interoperability is much easier to achieve at the interface level. CORBA objects supports specification-level interoperability. Figure 4.2 shows an example of how the object technology can simplify the development of advanced applications. Without the object support, all the I/O and network communications must be done through a set of basic data types. Adding an extra layer of object support, applications are transparent to the details of basic I/O and network communications. Other object support, e.g., object persistence, can be implemented on top of CORBA.

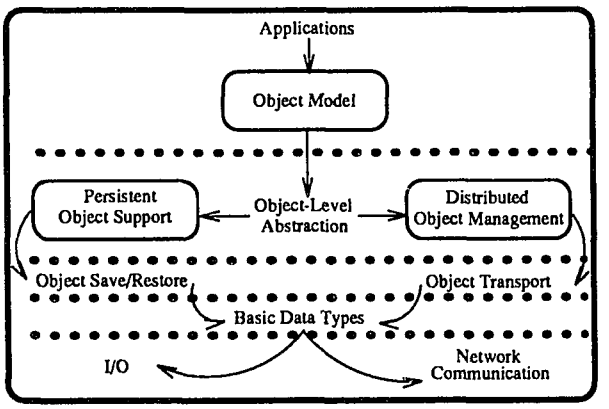

Figure 4.2: Object persistence and transport.

Runtime Support. The runtime behavior of *Zeus* views comes from a two-step mapping, view object $\mapsto$ CORBA object $\mapsto$ programming-level object. View objects are purely high-level abstractions. CORBA objects may encapsulate services, DBMSs, etc. Programming-level objects can be used in programming languages as program variables. The mappings between view objects and CORBA objects are handled by the $\mathcal{ZMS}$. The mappings between CORBA objects and programming-level objects are determined by a CORBA-compliant object request broker. At runtime, application programs rely on programming-level objects to perform the computation. CORBA objects have concrete runtime semantics which is provided in the forms of object invocation and method dispatching by CORBA-compliant object request brokers.

4.1.2 Computation Model (II)

In Chapter 3, we described the computation model, $\mathcal{CM}$, as a function determined by the multidatabase interface ($\mathcal{MI}$) and a set of database interfaces ($\mathcal{DI}$). A

global request is a parameter of $\mathcal{CM}$. In other words, $\mathcal{CM}$ accepts a global request, processes the request based on $\mathcal{MI}$ and $\mathcal{DI}$, and returns a result. Now that we have the runtime support from ORBs, the computation model can be expanded based on the two-step mapping. Suppose $\Lambda_{\mathcal{VT}}$ denotes the set of all view objects, $\Lambda_{\mathcal{C}}$ denotes the set of all CORBA objects, and $\Lambda_{\mathcal{S}}$ denotes the set of all system objects[1], then we can denote the object system of the $\mathcal{ZMS}$ as $\Lambda_{\mathcal{Z}}$, where $\Lambda_{\mathcal{Z}} = \Lambda_{\mathcal{VT}} \cup \Lambda_{\mathcal{C}} \cup \Lambda_{\mathcal{S}}$. The computation model, $\mathcal{CM}$, can be described as a sequence of mappings. First, $\mathcal{CM}$ accepts a global request which is associated with a subset of $\Lambda_{\mathcal{Z}}$. The $\mathcal{ZMS}$ creates a transaction that captures the semantics of the global request. $\mathcal{CM}$ maps the global request to a set of sub-requests against the CORBA objects based on $\mathcal{MI}$. Each of these sub-requests are then mapped to a set of database access routines. The $\mathcal{ZMS}$ updates the transaction to keep track of the associated sub-requests. As soon as the mappings are completed, the $\mathcal{ZMS}$ starts to execute the transaction. The information embedded in the transaction captures the precise semantics of the global request. We will elaborate on how the above mappings are materialized in Section 4.2.4.

4.1.3 Zeus Frameworks

We follow the definition of frameworks by Ralph Johnson [116] and define each of the *Zeus* framework as a set of classes. The refinement of the *Zeus* frameworks creates subframeworks which can also be deemed as the *Zeus* frameworks. Figure 4.3 shows the Zeus frameworks and how they are refined and specialized to create a running $\mathcal{ZMS}$. There are five top-level Zeus frameworks: *Zeus* framework ($\mathcal{F}_{\mathcal{Z}}$),

[1] System objects refer to the runtime $\mathcal{ZMS}$ components.

Database framework ($\mathcal{F}_D$), Integration framework ($\mathcal{F}_I$), Environment Framework ($\mathcal{F}_E$), and Domain framework ($\mathcal{F}_M$). We will elaborate on these frameworks when we describe the architectural design of the $\mathcal{ZMS}$.

Refinement. The five top-level $\mathcal{Z}$eus frameworks cover the high-level design of the $\mathcal{ZMS}$. These frameworks can be refined to form subframeworks that captures domain-specific details. For example, the database framework may be refined to a relational DBMS subframework and an object DBMS subframework based on the data model. Each subframework retains the common attributes inherited from the database framework and adds more attributes to cover model-specific features. The relational DBMS subframework can be further refined to a subframework for a specific RDBMS; e.g., Oracle. The subframework for Oracle can then be specialized to produce a class library for Oracle on UNIX-based platforms. The same refinement and specialization process can be done in a similar manner for other top-level frameworks and other subframeworks. It is clear that high-level frameworks are highly portable and reusable. Interoperability exists among low-level subframeworks and class libraries because they inherit the interfaces defined in high-level frameworks. The entire design is extensible since frameworks, subframeworks, and class libraries can be added at different levels of the design to accommodate changing requirements.

Assembly. In our framework-based architectural design, the low-level class libraries will contain the bulk of the $\mathcal{ZMS}$ code. Minimal coding is required to assemble a running $\mathcal{ZMS}$ for a specific permutation of computing environments and participating DBMSs. A running $\mathcal{ZMS}$ is an instance of the ZMSystem[2] class which

[2]We will elaborate on ZMSystem in Section 4.2.2.

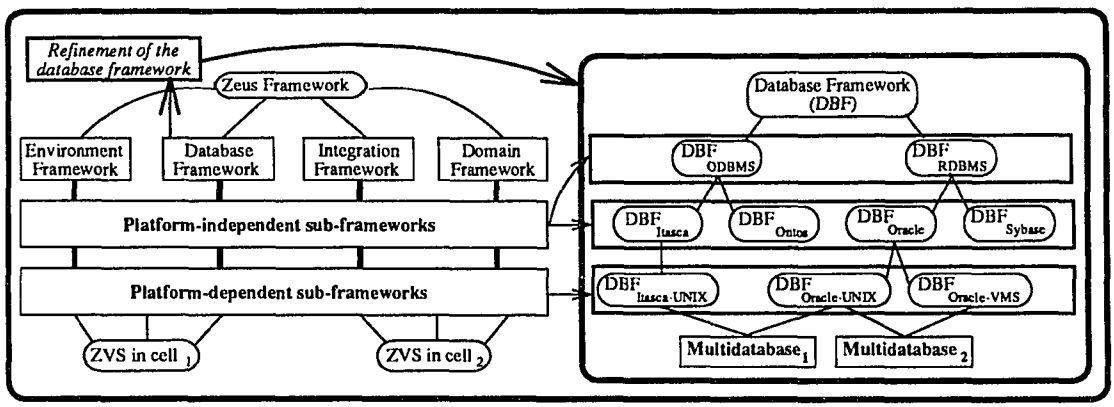

Figure 4.3: The *Zeus* frameworks.

defines the configuration of the $\mathcal{ZMS}$ and how the $\mathcal{ZMS}$ should be initialized followed by the invocation of other $\mathcal{ZMS}$ components. To assemble a running $\mathcal{ZMS}$, the low-level class libraries used should be identified and the ZMSystem object should be implemented. Like the approach described in [121], the *Z*eus frameworks provide two types of application program interfaces (APIs): a client API and a framework API. The client API allows the application programs to access the frameworks. The framework API allows the application programs to customized the code provided by the frameworks.

4.1.4 Client-Server Model

The $\mathcal{ZMS}$ has a client-server architecture. To facilitate the discussion of our view of a client-server model, we introduce the notations below to describe our view of a client-server model. A client-server system, $\mathcal{CS}$, can be modeled as a triple:

$$\mathcal{CS} = \langle \mathcal{R}_C, \mathcal{R}_S, \mathcal{R}_T \rangle, \mathcal{R}_T = \bigcup_i \eta_i(\hat{c}, \hat{s}), \text{ where}$$

$\mathcal{R}_C$ is a set of clients in $\mathcal{CS}$, $\mathcal{R}_S$ is a set of servers in $\mathcal{CS}$, $\mathcal{R}_r$ is a set of pair-wise relationships between clients and servers, $\hat{c} \in \mathcal{R}_C$, $\hat{s} \in \mathcal{R}_S$, and $\eta_i(\hat{c}, \hat{s})$ defines a set of request-response protocols between $\hat{c}$ and $\hat{s}$. It is possible that some components of $\mathcal{CS}$ can play the roles of both client and server. We describe a client as a runtime entity that request services from servers. On the other hand, a server is a runtime entity that processes the requests from clients. A request-response protocol has two parts: client-side protocol and server-side protocol. A client or a server may support multiple protocols. However, to initiate a conversation, a specific protocol must be identified as part of the client-server contract. Client-side or server-side protocols model the reactiveness of a client-server architecture; i.e., what messages clients or servers will respond and how. Now we can explain how the client-server model is applied to the $\mathcal{ZMS}$. We start with a discussion of **middleware**.

Middleware. Middleware is commonly referred to as the communication infrastructure that interconnects heterogeneous platforms and software systems [14]. In our project, middleware refers to the object request broker (ORB). The ORB provides transparent object transport, remote method invocation and multi-lingual support. Multiple object request brokers interoperate at the interface level. Using the ORB in our design avoids imposing strict requirements on participating systems since the interoperability requirements are embedded in the ORB.

Client-Server Architecture. Figure 4.4 shows a client-server architecture based on the use of ORBs. The ORBs provide an object layer on top of other network services; e.g., remote procedure calls, name service, etc. The architectural design of the $\mathcal{ZMS}$ results in a class hierarchy. Some of the instances of the classes

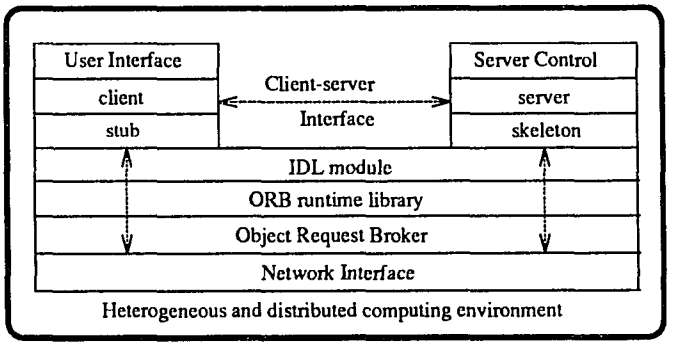

Figure 4.4: ORBs and Client-Server Architecture.

in this class hierarchy correspond to the $\mathcal{ZMS}$ runtime components which may be network servers or clients. The design of network servers has a number of dimensions: configuration, protocol, and architecture. The **configuration** dimension decides how a network server is initialized and identified, and how a service is bound to network servers. The **protocol** dimension describes how information is exchanged among network servers or between network servers and applications. The **architecture** dimension determines the overall structure of a network server that provides services and implements specified protocols. For example, a multi-service network server dispatches requests to other network servers, a concurrent network server can accept multiple client requests concurrently, etc. These design dimensions have been considered throughout the design of the $\mathcal{ZMS}$. We will explain what implementation issues are associated with these design dimensions later in this Chapter.

4.2 The Zeus Multidatabase System

Figure 4.5 gives an overview of the $\mathcal{ZMS}$. The logic of the $\mathcal{ZVM}$ is embedded in the server. The host manager and cooperating agent serve as the liaison between the

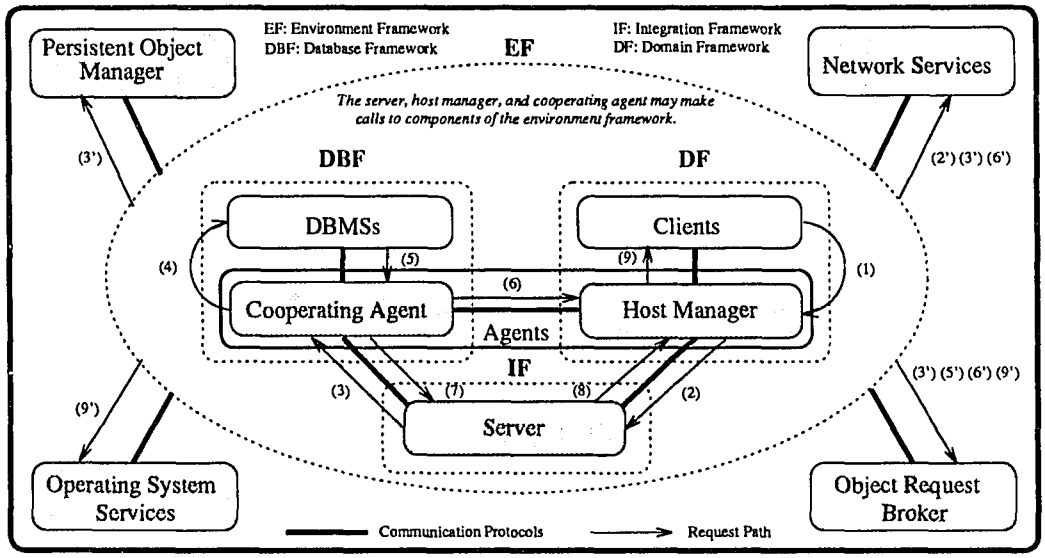

Figure 4.5: $\mathcal{ZMS}$ frameworks, major $\mathcal{ZMS}$ components, and a scenario.

server and the clients, and between the server and the DBMSs individually. Both the host manager and the cooperating agent are referred to as the $\mathcal{ZMS}$ agents. Through the server and the $\mathcal{ZMS}$ agents, the clients are able to access multiple, heterogeneous and distributed DBMSs transparently. To illustrate how the $\mathcal{ZMS}$ works, we provide an example and a global access scenario in Figures 4.6 and 4.5, respectively. Our design method achieves the following design criteria that we addressed in Chapter 1: portability, scalability, interoperability and extensibility. For the remainder of this section, we will describe the architectural design and major components of the $\mathcal{ZMS}$.

A Scenario. The numbered arrows in Figure 4.5 depict a global access scenario. The above scenario may be triggered by double-clicking "101 1990 Smith, Larry" in the graphical user interface provided by the $\mathcal{Z}$eus client in Figure 4.6. In step (1),

a global request is originated from a $\mathcal{ZMS}$ client and sent to the host manager. The host manager finds an available $\mathcal{ZMS}$ server via the network services (step (2')), and routes the request to the server in step (2). The server interprets the request by interacting with the persistent object manager and maps view objects to run-time objects and threads by contacting the object request broker in step (3'). In step (3), the server finds available cooperating agents to send the requests spawned from the threads generated in step (3'). The cooperating agent routes the request to the DBMS in step (4) and the DBMS responds to the request in step (5). Suppose that the cooperating agent has received data from the DBMS, the data are sent to the requesting host manager in step (6) and at the same time a message is sent to the server in step (7) indicating that the request has been served on behalf of the thread. Step (6') shows that the data transfer may require interactions with network services and the object request broker. The server's transaction manager determines whether the transactions related to the original request should be committed. When the transaction is committed, the server sends a message to the host manager in step (8). The host manager makes the data visible to the client in step (9).

4.2.1 Architectural Overview

Now, we can depict a novel architecture for the $\mathcal{ZMS}$. The $\mathcal{ZMS}$ architecture is composed of a kernel and a set of interface components. The $\mathcal{ZMS}$ kernel appears within a circle in Figure 4.7. The $\mathcal{ZMS}$ kernel provides the major functionality of the $\mathcal{ZMS}$. The kernel is integrated with the local computing environment via a set of interface components: the Common Access Interface, the $\mathcal{ZMS}$ Resources Interface, the Network Services Interface and the Cooperating DBMS Interface. The Common

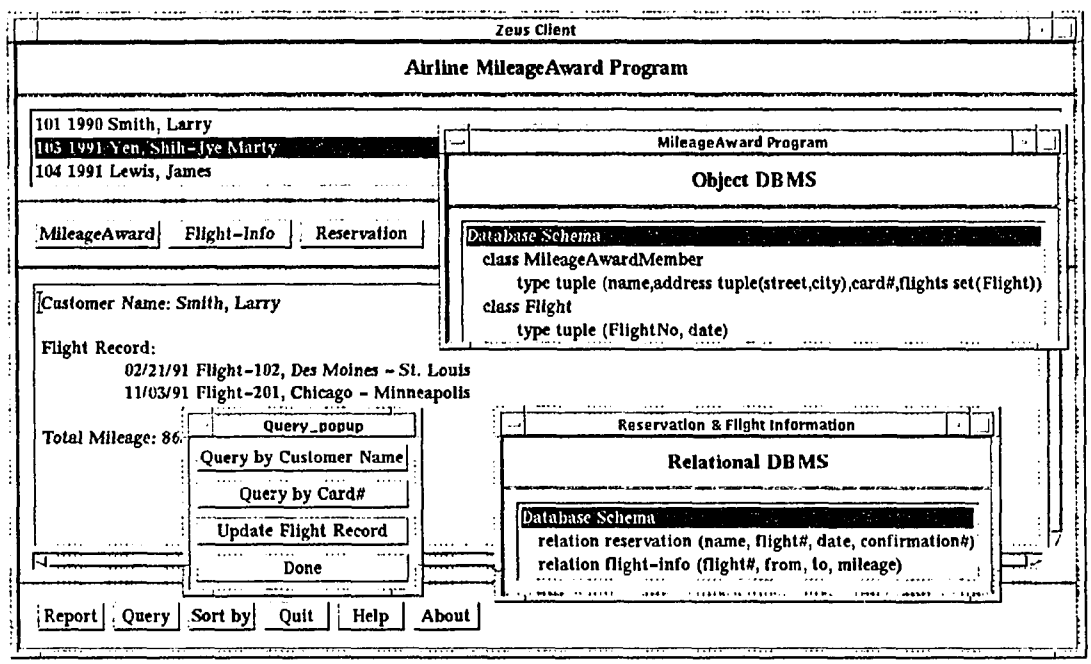

Figure 4.6: The database schema of the running example and a sample Zeus client.

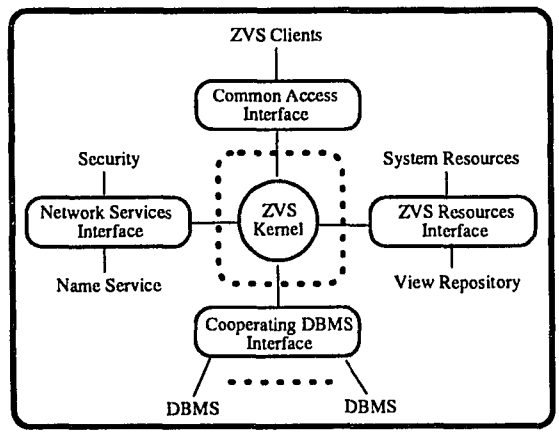

Figure 4.7: The $\mathcal{ZMS}$ architecture.

Access Interface allows the $\mathcal{ZMS}$ clients to access $\mathcal{ZMS}$ resources or the data from participating DBMSs, e.g., a $\mathcal{ZMS}$ client may accept a request to browse existing views. The $\mathcal{ZMS}$ Resources Interface provides management interfaces between the kernel and the $\mathcal{ZMS}$ resources. The Network Services Interface links the kernel to local or remote hosts. The Cooperating DBMS Interface provides the link between the kernel and participating DBMSs in the way that the $\mathcal{ZMS}$ can coordinate the requests and responses between the $\mathcal{ZMS}$ clients and the DBMSs.

We start the architectural design of the $\mathcal{ZMS}$ by identifying the objects and classes that model the functionality and requirements of the $\mathcal{ZMS}$. These objects and classes are later mapped to major components of the $\mathcal{ZMS}$. The relationships and responsibilities of the major $\mathcal{ZMS}$ components are captured by the class hierarchies and behavioral semantics of the methods defined for $\mathcal{ZMS}$ classes. Major components of the $\mathcal{ZMS}$ are shown in the shaded areas of Figure 4.5. The architectural design of the $\mathcal{ZMS}$ is based on a class hierarchy, $\mathcal{H} = \mathcal{A} \cup \mathcal{C}$, where $\mathcal{A}$ is a set of abstract classes, and $\mathcal{C}$ is a set of concrete classes partially ordered by a **class-subclass**

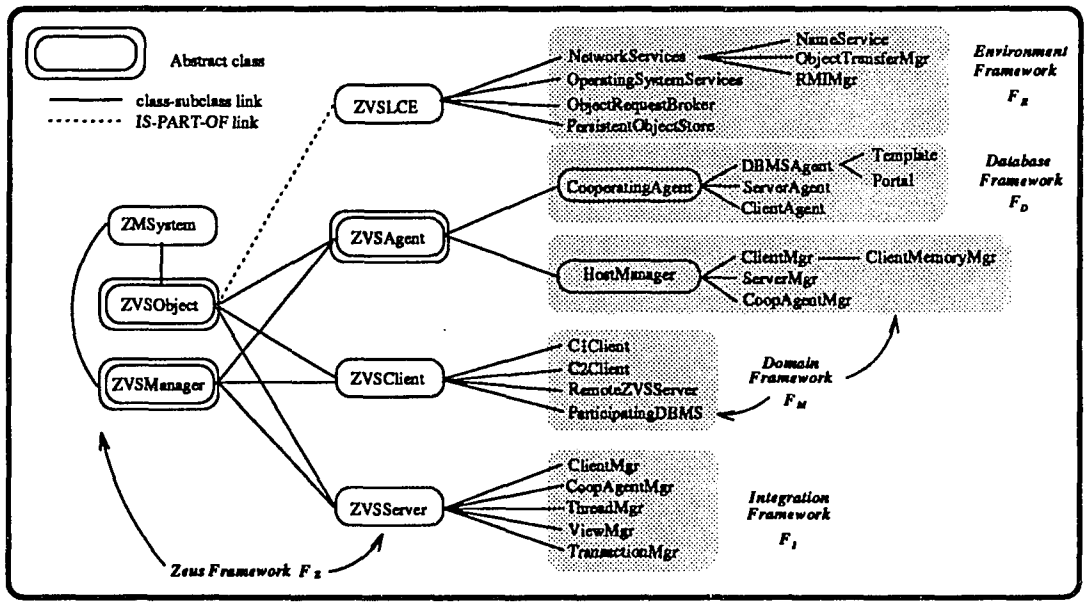

Figure 4.8: Part of the $\mathcal{ZMS}$ Class Hierarchy.

relationship. Part of $\mathcal{H}$, i.e., $\mathcal{H}'$, that corresponds to the high-level design of the $\mathcal{ZMS}$ is shown in Figure 4.8. $\mathcal{H}'$ is partitioned into five disjoint sets of classes where each set forms one of the five top-level $\mathcal{Z}$eus frameworks; labeled $\mathcal{F}_Z, \mathcal{F}_E, \mathcal{F}_D, \mathcal{F}_M$, and $\mathcal{F}_I$ in Figure 4.8. These five $\mathcal{Z}$eus frameworks are refined, and finally assembled and instantiated to create a runtime $\mathcal{ZMS}$.

To guarantee that the notion of frameworks will be useful to the development of the $\mathcal{ZMS}$, we impose design constraints on the properties of $\mathcal{Z}$eus frameworks. We define these constraints by modeling the set of all the $\mathcal{Z}$eus frameworks as a directed acyclic graph. The framework-based design consists of two steps: the refinement of frameworks and the assembling of the runtime system. The design process spawns a set of design frameworks. We model these design frameworks by a directed graph G. A graph $G = (X, U)$ is composed of:

1. A finite set $X = \{x_1, x_2, ..., x_n\}$, the elements of which are called nodes. Each node corresponds to a design framework. The set of subscripts of $x \in X$ is denoted as N_X. We refer to a node that is neither a root node or a leaf node as an interior node.

2. A subset U of the Cartesian product $X \times X$, the elements of which are called arcs. Each arc corresponds to the relationship between a parent framework and a subframework. A subframework inherits all the classes of its parent framework. In other words, the design embedded in a parent framework is fully reusable by its subframework. An arc from x_1 to x_2 represents an inheritance relationship.

Different graphs correspond to different design alternatives. One of the goals of our approach is to identify automatable ways of analyzing framework graphs and to provide guidance for designers to choose a good and possibly the best design. The criteria for a good design are based on software metrics and whether or not the design criteria are met. The structure of framework graphs mirrors the refinement of frameworks and reflects the domain knowledge. We have developed a set of constraints that restrict the structure of the graph in order to filter out favorable design alternatives. Constraints A and B are given below as an example. Figure 4.9 shows some of the possible patterns that may exist in a framework graph and its subgraphs.

Constraint A: *A source in G must have more than one subframeworks, a sink in G must have siblings and an interior node in G must have siblings and more than one subframework.* $\diamond$

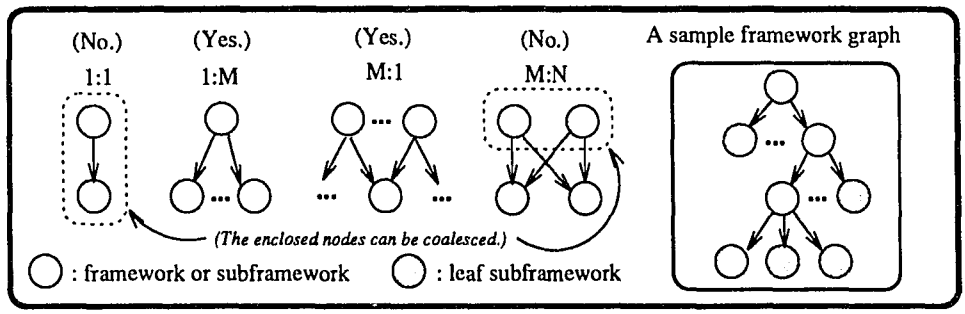

Figure 4.9: Possible relationships between parent frameworks and subframeworks.

Constraint B: *Suppose G is a framework graph, the relationship between a parent framework and a subframework in G can be one-to-many or many-to-one. Many-to-many relationships may exist if and only if G satisfies Constraint A and the set of arcs (A_P) between a set of parent frameworks (P) and the set of subframeworks (S) associated with P is a strict subset of $P \times S$; i.e., $A_P \subset P \times S$.* $\diamond$

A source in G must have more than one subframework. Otherwise, the source can be coalesced with its only subframework. In other words, a sink must have siblings and an interior node must have siblings and more than one subframework. This is why we need Constraint A. If $A_P = P \times S$, then all the frameworks in P can be coalesced into one parent framework. The semantics of Constraint B can be interpreted as a way to keep the design concise and increase the reuse efficiency while avoiding redundant work.

For the remainder of this Chapter, we describe major $\mathcal{ZMS}$ components in two aspects: modeling and architecture. The modeling of an $\mathcal{ZMS}$ component identifies

the classes and objects as well as their associated responsibilities for meeting the functionality and requirements of the $\mathcal{ZMS}$ component. The relationships among classes and objects, and how the design is mapped to the client-server model are captured in the description of the architecture of the component.

4.2.2 Top-Level ZMS Design

We start the discussion of the $\mathcal{ZMS}$ architectural design by identifying the seven top-level classes that form the $\mathcal{Z}$eus framework, $\mathcal{F}_Z$; i.e., ZVSObject, ZVSServer, ZVSystem, ZVSClient, ZVSLCE, ZVSAgent, and ZVSManager. ZVSObject is an abstract class. All system classes are subclasses of ZVSObject except ZVSLCE. We use ZVSObject to group common attributes and operations of all system objects. For example, ZVSObject has an attribute that links to services of the computing environment. Subclasses inherit this attribute from ZVSObject and thus all objects possess a link to their own computing environment. ZVSObject also has two common operations, startup and shutdown, that coordinate the initialization and termination of services offered by the individual $\mathcal{ZMS}$ components.

Runtime System. A runtime $\mathcal{ZMS}$, $\mathcal{P}$, is created by linking an instance of ZMSystem with a set of class libraries. The major functionality of the $\mathcal{ZMS}$ system object is to coordinate the start-up and shutdown sequences for major $\mathcal{ZMS}$ components including the $\mathcal{ZMS}$ server, the $\mathcal{ZMS}$ clients, and the $\mathcal{ZMS}$ agents. The the $\mathcal{ZMS}$ server is an instance of ZVSServer. Instances of ZVSClient correspond to the $\mathcal{ZMS}$ clients. ZVSClient defines the communication protocol between the $\mathcal{ZMS}$ clients and other $\mathcal{ZMS}$ components. ZVSAgent is an abstract class that contains the

common attributes and properties of the $\mathcal{ZMS}$ agents.

Network Servers. ZVSManager is an abstract class that provides a template for modeling the $\mathcal{ZMS}$ network servers in three dimensions: protocol, service, and configuration. Each network server inherits a command table attribute that contains the syntax followed by clients to communicate with the network server. Multi-service network servers inherit a Dispatch operation that handles the dispatching of requests to appropriate subordinate network servers. Each network server also inherits a Configuration attribute that contains the configuration of the network server.

Architecture. The $\mathcal{ZMS}$ has a client-server architecture. The client-server relationship may exist between any pair of the $\mathcal{ZMS}$ components which spread over a distributed network of hosts. The client-server relationship is described by the behavioral semantics bound to the specifications written in IDL. To bring the entire $\mathcal{ZMS}$ up and running, a runtime $\mathcal{ZMS}$ system must be started first. Then, the network servers of the $\mathcal{ZMS}$ are started. A normal shutdown of the $\mathcal{ZMS}$ starts with the shutdown of the $\mathcal{ZMS}$ network servers. The $\mathcal{ZMS}$ runtime system shuts down after the shutdown of all the $\mathcal{ZMS}$ network servers is completed.

4.2.3 ZMS Environment

The $\mathcal{ZMS}$ Environment is modeled by the environment framework ($\mathcal{F}_E$), which includes four major components of the surrounding computing environment: network services, operating system services, the object request broker, and the persistent object store. Since the $\mathcal{ZMS}$ components may run in different computing environments;

e.g., different platforms, different operating systems, etc., an instance of ZVSLCE explicitly determines the computing environment within which the $\mathcal{ZMS}$ component is running. $\mathcal{F}_E$ describes the high-level interface specifications of the services offered by a ZVSLCE object. The implementations of these services may vary with different computing environments. A refinement of $\mathcal{F}_E$ will spawn the matching subframeworks and class libraries that capture the specialized features of a specific computing environment. However, all $\mathcal{ZMS}$ components still have a uniform interface to external services offered by a ZVSLCE object due to the common high-level interface specifications.

Architecture. The design of the $\mathcal{ZMS}$ relies on interoperable object request brokers [118, 122] to provide a distributed object infrastructure. Such an infrastructure along with persistent object stores allow us to apply the object concept and modeling power at different levels of the $\mathcal{ZMS}$ development. We require that all hosts that run the $\mathcal{ZMS}$ components must be running a CORBA-compliant object request broker and must have access to a persistent object store. Therefore, the object transfer and remote method invocation can be done through the interoperation among the object request brokers and persistent object stores running on participating hosts. Network and operating systems services can be encapsulated in CORBA objects where interface specifications are bound to actual implementations.

4.2.4 ZMS Server and ZMS Client

The $\mathcal{ZMS}$ server interacts with the $\mathcal{ZMS}$ agents to receive requests from clients, interpret the request, and generate requests to retrieve data on behalf of the original

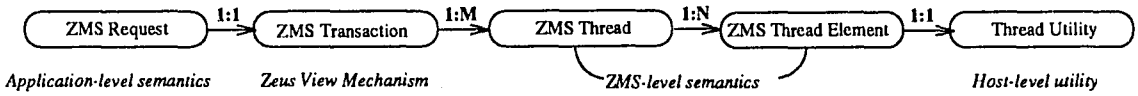

Figure 4.10: Request, transaction, and thread in the $\mathcal{ZVS}$.

request. In our design, the data are sent directly to the requesting agents and the $\mathcal{ZMS}$ server coordinates the commitment of transactions resulting from the request. The $\mathcal{ZMS}$ server, ZVSServer, and instances of its subclasses implements the $\mathcal{ZVM}$. The $\mathcal{ZMS}$ server is modeled as a multi-service server that has multiple managers. The $\mathcal{ZMS}$ server along with its managers implement a computation model that supports threads and transactions.

Request, Thread, and Transaction. A thread is a single sequential flow of control. The thread utility of an operating system allows multiple lightweight threads to run concurrently within a single address space. As can be seen in Figure 4.10, application logic is embedded in $\mathcal{ZMS}$ requests. The $\mathcal{ZVM}$ creates corresponding transactions for certain types of the $\mathcal{ZMS}$ requests. A $\mathcal{ZMS}$ transaction is the $\mathcal{ZMS}$'s view of the associated application logic. Multiple threads may be spawned from a $\mathcal{ZMS}$ transaction. A $\mathcal{ZMS}$ thread corresponds to a task identified by the $\mathcal{ZMS}$. A thread may in turn generate multiple thread elements that cooperate to complete a task. A thread element is implemented by the thread utility or operating system utilities. There is a one-to-one correspondence between the thread element and the thread supported by an operating system.

Computation Model. The computation model defines how the $\mathcal{ZMS}$ interprets the request and coordinates the handling of data; i.e., the runtime seman-

tics of the $\mathcal{ZMS}$. We require that all the $\mathcal{ZMS}$ components have a contacting CORBA-compliant ORB in order to support the two-step mapping described in Section 4.1.2. The computation model described in Section 4.1.2 is implemented by the notions of requests, transactions, and threads. The threads are used to implement sub-requests. We use a thread element to further implement a thread by the thread utility of the hosting operating system. Therefore, the computation model is implemented by the $\mathcal{ZMS}$ components defined in ZMSServer, ViewMgr, TransactionMgr, and ThreadMgr.

$\mathcal{ZMS}$ Client. The $\mathcal{ZMS}$ client refers to any runtime entity that is external to the $\mathcal{ZMS}$ and interacts with the $\mathcal{ZMS}$ components. A $\mathcal{ZMS}$ client can be a C1 $\mathcal{ZMS}$ client, a C2 $\mathcal{ZMS}$ client, a program that interacts with a participating DBMS, or a remote $\mathcal{ZMS}$ server. C1 $\mathcal{ZMS}$ clients provide services for the management of views including the creation, deletion, update, browsing and installation of views in the view repository. C2 $\mathcal{ZMS}$ clients provide services for users to access global resources through views.

Architecture. Figure 4.11 shows two IDL source files that specify ZVSManager and ZVSServer. The syntax of IDL is similar to that of C++ except that IDL is purely declarative. The interface definition of ZVSManager specifies the communication protocol for network servers. The header files, ZVSRequest.idl and ZVSLCE.idl, are included in ZVSManager.idl since both ZVSRequest and ZVSLCE.idl are used in the definition of ZVSManager. The specifications defined in ZVSManager are acquired by ZVSServer through inheritance. A ZVSServer object has attributes linked to its managers which are all network servers and have back pointers linked to the ZVSServer

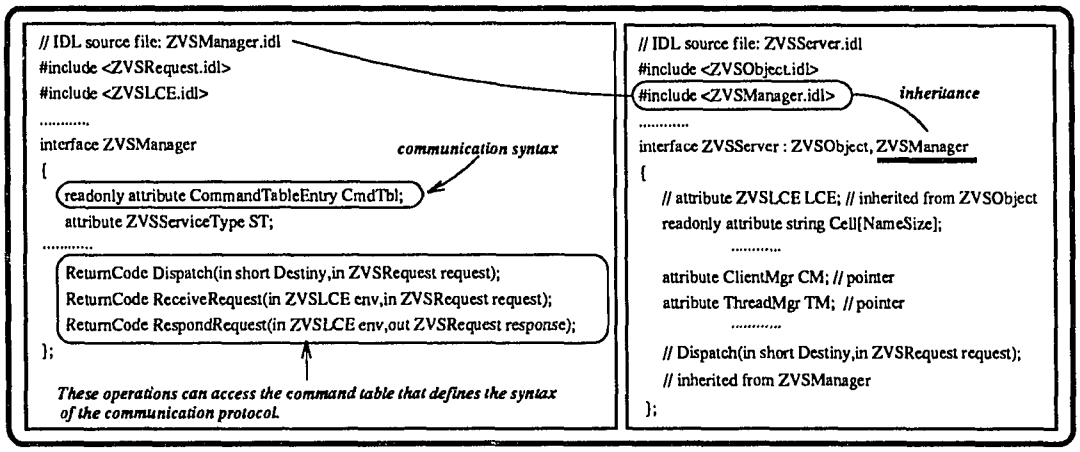

Figure 4.11: IDL examples.

object. Therefore, the $\mathcal{ZMS}$ server and its associated managers do not have to run on the same host. Each of these network servers may have its own syntax and implementation of their communication protocols through different bindings of the IDL interface specifications of the $\mathcal{ZMS}$ to the actual implementation. As can be seen in Figure 4.11, each function appeared in the IDL source files has a return type, a function name, and a signature. The implementation of each function appears in an expanded template file resulting from compiling the associated IDL source file.

4.2.5 ZMS Agent

The $\mathcal{ZMS}$ agents are either host managers or cooperating agents. The host manager establishes and maintains the communication links between the $\mathcal{ZMS}$ clients and the $\mathcal{ZMS}$ server, or between the Zeus clients and the cooperating agents. The $\mathcal{ZMS}$ clients may send two types of requests to the $\mathcal{ZMS}$ server via the host manager: a request for global data and a request for information about global data. The host manager sends the requests to the server and continues to coordinate the con-

versation that follows. As soon as the data have been received by the host manager from the server or the cooperating agents, the host manager interacts with the $\mathcal{ZMS}$ server to decide whether the data needs to be processed and when the data should be made visible to the clients. Whether the data need to be processed depends on the application semantics which should be part of the view definition. When the data should be made available to the client depends on the transaction management of the $\mathcal{ZMS}$ server. When the data are ready, the host manager puts the data in the virtual memory space of the client processes.

The cooperating agent serves as an agent between the $\mathcal{ZMS}$ clients and one or more DBMSs. The request for data received by the cooperating agent is dispatched from the server. The cooperating agent interacts with the DBMSs to retrieve the data, processes the data if necessary, sends the data to the requesting host manager, and notifies the server that the request has been processed.

Template and Portal. We have generalized the notions of templates and portals developed in the OR Interface [90]. In the $\mathcal{ZMS}$, a template models a generic interface that can be used to automate the generation of the DBMS access routines, i.e. portals, based upon a user-defined Zeus view. The result is a customized interface to the DBMS tailored to the requirements of a specific application. Templates and portals are modeled as instances of Template and Portal which are subclasses of DBMSAgent.

Architecture. Host managers, cooperating agents, and their associated managers are distributed in a network of hosts. Instances of ClientMgr, ServerMgr, and CoopAgentMgr run on the same host as that of the HostManager object. The

ClientMemoryMgr object runs on the same host where the $\mathcal{ZMS}$ client is running. Since participating DBMSs may not have the capability of running as a network server, we use ParticipatingDBMS to model the way the participating DBMSs are expected to work with the $\mathcal{ZMS}$. A ParticipatingDBMS object runs on the same host as that of a participating DBMS and serves as an intermediate network server between the $\mathcal{ZMS}$ and the DBMS. A database access routine, i.e., a portal, does not have to run on the same host as that of the accessed DBMS. Portals inherit from DBMSAgent the communication protocol to interact with a ParticipatingDBMS object.

4.2.6 Specification

We use CORBA IDL to specify the high-level design of the $\mathcal{ZMS}$. A listing of this specification in the form of IDL source files is included in Appendix B. Along with each IDL source file, we provide descriptions on the behavioral semantics associated with the interface defined in the IDL source file.

4.3 Implementation Issues

During the course of experimenting with developing part of the $\mathcal{ZMS}$, we have uncovered a number of implementation issues that have contributed to our established experience and will help us in the implementation of a full-blown multidatabase system. Our target computing environment for implementing the $\mathcal{ZMS}$ is composed of a local area network located in the ISU Computer Science Department, an AS/400 and a large-scale, campus-wide distributed computing environment called Project Vincent. The backbone of the campus network is Digital Equipment Corporation's

implementation of the Fiber Distributed Data Interface (FDDI) that provides a data transmission speed of 100 megabits per second. Multiprotocol routers and Ethernet bridges were used to connect Ethernet segments of campus buildings to the FDDI backbone. Campus connection to NSFnet has T1 speed (1.5 megabits per second). Several high-performance workstations are directly attached to the FDDI nodes.

The major network services in Project Vincent were originally ported from Project Athena [30] and have undergone continuing revision since then. There are four basic types of servers under Project Vincent: Read-Only, System Management Services, Kerberos, and Full-Service servers. The Read-Only server maintains a repository for operating system softwares and other shared softwares. The Kerberos server provides authentication services to the entire network. The System Management Services servers provide management facilities for the entire distributed system. The Full-Service servers provide most of the remaining user-accessible services including distributed file storage (NFS and AFS), Zephyr message delivery, Hesiod name service, and so on. NCS RPC [68] is supported in Project Vincent. The Computer Science local area network runs NFS and has TCP/IP connections to Project Vincent. The QTCP subsystem on the AS/400 provides TCP/IP connections to both Project Vincent and the Computer Science local area network.

Based on the above computing environment, a number of implementation issues have been explored in Project *Zeus*. We view this experience as valuable guidance for the next phase of Project *Zeus*; i.e., the completion of a working multidatabase system based on the $\mathcal{ZVM}$ and the methodology that we described earlier in this Chapter.

4.3.1 Network Servers

There are a number of issues related to the implementation of network servers. The first issue is the choice of a communication suite that provides naming, addressing, and the services of a transport protocol. We are also concerned about how the local and remote interprocess communications (IPCs) offered by the host operating system interface are integrated with the communication suite. To support the notion of the $\mathcal{ZMS}$ thread, we need advanced process management fcailities from the host operating system. Other issues like security, the transfer of large objects and the external data representation are all important in the implementation of network servers.

4.3.2 The AS/400 and Project Zeus

By choosing the AS/400 as one of our prototyping platforms, we are able to increase the heterogeneity of the computing environment. There is a number of critical aspects that distinguishes the AS/400 from UNIX-based platforms. In the context of machine architecture and operating system, the layered organization of the function in the AS/400 consists of five layers. The bottom three layers (the hardware, the horizontal licensed internal code (HLIC) and the vertical licensed internal code (VLIC)) implement the machine interface (MI). The next layer is the operating system and the top layer consists of high level language compilers and applications. In the context of database support, the database component of the AS/400 is fully integrated. Database functions are implemented in the operating system and the MI. As a result, most of the function that is supported by the database can be accessed through the use of the systems command language (CL). One peculiar characteristic

of the AS/400 is the notion of single-level store and object paradigm. Everything is stored on the AS/400 as an object. Objects are encapsulated through the use of the abstract data paradigm. The methods defined for system objects are defined by the MI instruction set. The code which implements all of these methods is contained within the VLIC. The $\mathcal{ZVS}$ design separates all these AS/400-specific features from system implementations through portable interface definitions. This separation will ensure a smooth transition to IBM's implementation of the CORBA-compliant ORB on the AS/400 [130] as it becomes available.

4.3.3 Programming with CORBA

Partly due to the fact that the implementation of commercial object request brokers is still at its infancy, the reported experience with CORBA has been scarce, especially for complex and large-scale software systems. The adoption of CORBA in Project *Zeus* has helped us simplify the design of the $\mathcal{ZMS}$. After gaining the programming experience with CORBA and experimenting with alternative $\mathcal{ZMS}$ designs, we expect that the bulk of the system source will be significantly smaller compared to the same design developed without CORBA.

Modeling and Abstraction. Part 1 of Figure 4.12 shows the three development stages separated by dotted lines. The system design started out with an application object model. We use object-oriented state machines (OOSMs) to capture the reactiveness of the $\mathcal{ZVS}$. The behavioral semantics is described in formal specifications. As soon as the system is well-defined, we started to code our design in the IDL files. Part 2 of Figure 4.12 lists a sample IDL source file. The IDL source files

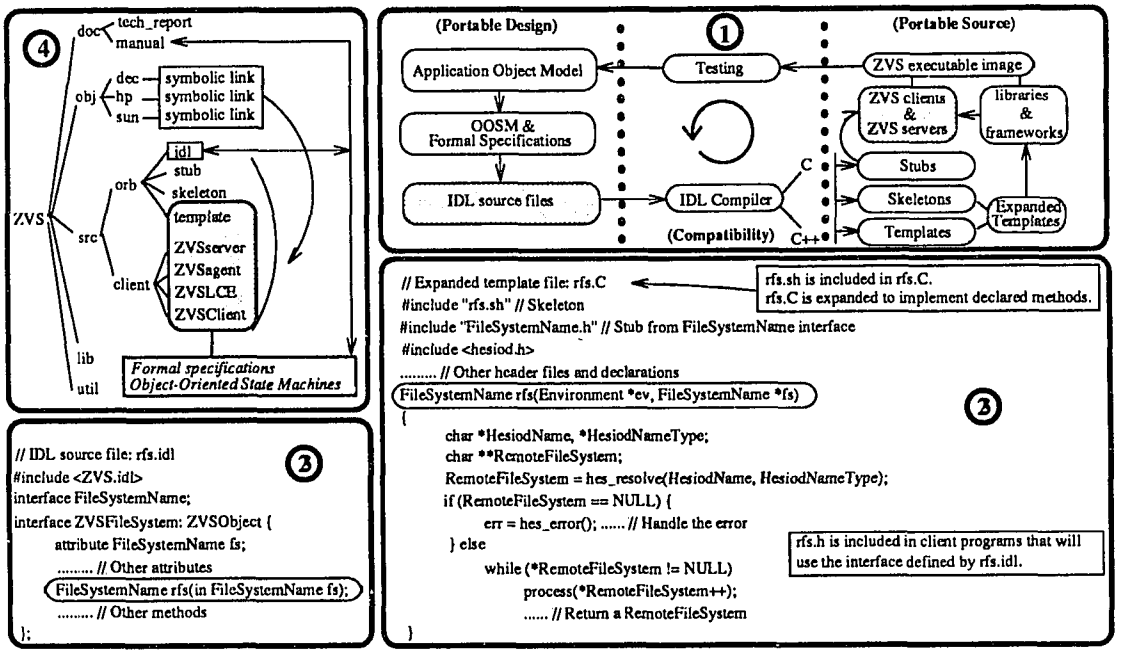

Figure 4.12: Programming with CORBA.

only cover the application object model. The interactions among major components and behavioral semantics are embedded in the implementation. In other words, the full-blown system is made of the IDL source files, $\mathcal{ZMS}$ clients, $\mathcal{ZMS}$ servers, and expanded templates.

Development. The IDL source files are then compiled to generate stubs, skeletons, and templates for the programming languages that have IDL mappings. Each template has the specifications of operations defined in IDL files. The templates need to be expanded to include the implementations of the declared operations. The templates include most of the system behavioral semantics. Part 3 of Figure 4.12 lists the expanded template that includes the implementation of the method, ZVSFileSystem, defined in rfs.idl. The expanded templates are compiled and archived

into class libraries or grouped as frameworks. Now, we can write client programs that use the objects and operations defined in IDL files. These client programs are linked with libraries to create the $\mathcal{ZMS}$ executable images. The interactions among system components are embedded in client programs.

Testing and Maintenance. Component tests can then be done for each client program. In case the design has changed, we can follow the same cycle to update the specifications and implementations. As can be seen, the first stage is focused on the design. The methodology we used is independent of any programming languages or platforms. The entire design is portable. Whether the files generated from the IDL compiler are reusable in other CORBA implementations depends on the compatibility of IDL compilers. Cautions need to be taken to prevent the IDL specifications and templates from getting out of sync. The resulting source configuration is shown in part 4 of Figure 4.12. We keep a single copy of the source for the three major UNIX-based platforms to enhance the portability and reduce the maintenance cost of the $\mathcal{ZMS}$.

4.3.4 Global Architecture

The implementation of the $\mathcal{ZMS}$ depends on the hosting computing environment. We want to separate the design of the $\mathcal{ZMS}$ from the implementation details that depend on the hosting computing environment. How this separation has been achieved can be explained through a global $\mathcal{ZMS}$ architecture. We group the interface components into a set of CORBA objects called LCE. This allows us to specify a portable interface between the $\mathcal{ZMS}$ kernel and the hosting computing environ-

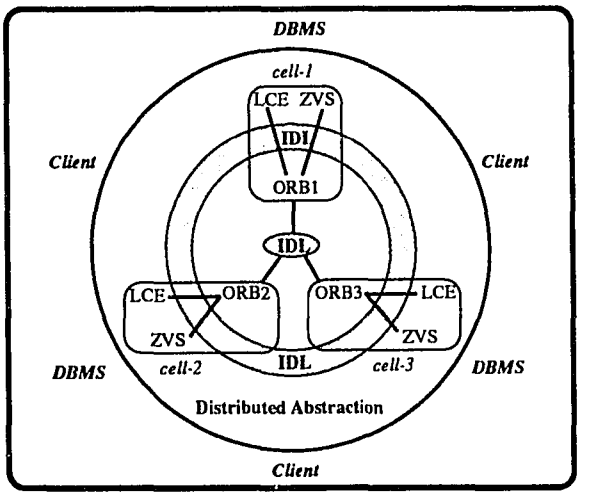

Figure 4.13: A global $\mathcal{ZVS}$ architecture.

ment. Now we can picture our design in a more global context. Figure 4.13 shows the simplified architecture of the $\mathcal{ZMS}$ community. A cell may refer to a campus computing environemnt, an organizational computing system, etc. Within a cell, the $\mathcal{ZMS}$, LCE, and ORB interact with one another through IDL modules. Between the cells, the ORBs communicate with each other through CORBA objects which are also defined in IDL modules. In Figure 4.13, the shaded ring shows that the $\mathcal{ZVS}$ and LCE of all cells may share the same IDL specification to interact with each other. This achieves design portability of the $\mathcal{ZMS}$ across different computing environments. The shaded ellipse shows that the ORBs of all cells share the same IDL specification for interactions among ORBs. This achieves specification-level interoperability among ORBs. From clients' point of view, there is a single uniform interface for access to multiple DBMSs. This is how the distributed abstraction is achieved in the $\mathcal{ZMS}$.

4.3.5 Performance Optimization

Like other multidatabase systems, the $\mathcal{ZMS}$ provides an extra layer between applications and multiple DBMSs. The implementation and functions of such a software layer will impose extra performance cost as opposed to direct coupling of native DBMSs. Despite the extensive study of multidatabase issues, there were few guidelines to follow in Project *Zeus* to accommodate performance optimization. Instead, we have made a number of decisions as the project progressed. First of all, we decided to separate specifications from implementations. A high-level abstraction can be easily optimized by virtue of its declarative nature. Secondly, we chose to support different degrees of integration. Partial integration allows the system to scale-up and reduces the cost of maintaining the integrated schema. Total integration reduces the levels of integration. Finally, we chose to allow different levels of global access. For example, a user tries to access a local DBMS through the $\mathcal{ZMS}$. The $\mathcal{ZMS}$ may intelligently establish the local connection instead of moving the data between the $\mathcal{ZMS}$ server host and user's host. The same kind of intelligence may be used to minimize the movement of objects through network communications.

CHAPTER 5. FUTURE DIRECTIONS AND SUMMARY

In this dissertation we described an innovative approach to multidatabase integration and interoperation. We discuss our future research directions that will expand on the work we presented here. We also summarize the major components of our solution and conclude with a description of the prospects of Project *Zeus*.

5.1 Future Directions

There are a number of directions for future research to expand on the work presented in this dissertation and to explore new applications that fit well with the *ZVS*.

5.1.1 Larch-Style Formal Specifications

In computer systems development, formal methods provide a way for specifying, developing, and verifying systems. In Project *Zeus*, we chose to use formal methods to specify the *ZMS* for four reasons. First of all, the *ZMS* is much more complicated than other distributed systems in terms of the system design and implementation. The increasing complexity demands an implementation-independent specification tool. Secondly, we used CORBA/IDL to specify the *ZMS*. Since CORBA/IDL is purely declarative, the reactiveness and behavioral semantics are

not captured in IDL source files. We need other specification tools to describe the reactiveness and behavioral semantics of the $\mathcal{ZMS}$. Thirdly, CORBA/IDL has mappings for multiple programming languages. A complex system is likely to be written in a number of programming languages. Using a common syntax to describe the system improves the maintenance and readability of the system design and implementation. Finally, formal specifications have a solid mathematical ground and are inherently more concise than code. This will simplify the port of the $\mathcal{ZMS}$. An adapted Larch-style formal specification language [58], Larch/CORBA, is being investigated at Iowa State University. As Project *Zeus* enters its next phase, we plan to use Larch/CORBA to specify the behavioral semantics of IDL source files as well as the reactiveness of the $\mathcal{ZMS}$ components.

5.1.2 Multidatabase Management Facilities

We plan to add two major multidatabase management facilities to the $\mathcal{ZMS}$: a query interface and a multidatabase transaction management facility. We have already started to investigate the use of an object query language as a query interface to the $\mathcal{ZMS}$. The functionality of such a **multidatabase query interface** has three major aspects. First of all, the query interface should allow users to query meta data; i.e., the information about the data that are available. In our case, such information is embedded in views stored in view repositories. Secondly, the query interface should allow users to transparently access global data from multiple information sources. Thirdly, the query interface must have the capability of dealing with multimedia data types which have started to appear in more and more applications.

The **transaction management facility** will provide a mechanism to comple-

ment the notion of $\mathcal{ZMS}$ transactions. We need to work on two major issues. First of all, we must detail the representations of the $\mathcal{ZMS}$ transactions and their mappings to the query and view constructs. Secondly, we must deal with the semantics of multidatabase transactions under the assumptions that participating DBMSs are heterogeneous, autonomous, and distributed.

5.1.3 An Open and Cooperative Framework

Emerging software technologies, such as object-oriented technology, apply new concepts and methodologies to achieve code reuse, tool integration and database support for CASE within an object-oriented framework. The concept of component-oriented software development [82] shifts the development of large-scale distributed applications to the composition of plug-compatible software components. Megaprogramming can be deemed as an extension to component-oriented software development. This extension will promote large-scale, cross-platform, cooperative and concurrent programming in distributed computing environments. A discussion of research issues in megaprogramming can be found in [112]. Recent advances in both data and software engineering exhibit a number of common characteristics; i.e., large-scale, wide-area, increased distribution, heterogeneity, cooperation and interoperation. There is also a close tie between these two areas. For example, a megaprogram [112] may need integrated access to several information systems. An information system may provide management facilities for inter-related software components. Since the very beginning of Project *Zeus*, we have decided to let the design of the $\mathcal{ZVS}$ remain an open architecture which is extensible and adjustable to changing technologies. The long-term goal of Project *Zeus* is geared toward supporting

integration and interoperation of resources in a cooperative fashion. Ultimately, the $\mathcal{ZVS}$ will amalgamate future data and software engineering applications in an open and cooperative framework.

5.2 Summary

We have presented a view mechanism, the $\mathcal{ZVM}$, that shields underlying heterogeneity, complexity and distribution from the developers of future distributed applications. This transparency is provided through encapsulation, semantic relativism, and distributed abstraction modeling. The interaction between *Zeus* views, the abstraction and semantics embedded in *Zeus* views and the construction of interoperable *Zeus* views provide the basis for the integration and interoperation of global services and resources at the application level. The $\mathcal{ZVM}$ provides a unique solution for solving the modeling problem that we addressed in Chapter 1. The framework-based design method provides a way to control the complexity of the system design and achieve the design criteria. This is our solution to the system design problem.

At early stages of Project *Zeus*, we developed the preliminary $\mathcal{ZVS}$ design based on the specification of a distributed object infrastructure described in CORBA [122]. CORBA-compliant object request brokers provide the runtime support for the computation model of the $\mathcal{ZVS}$. We also applied our approach to provide a solution for multidatabase issues. The integration via *Zeus* views was successfully integrated into our earlier design. Our continued investigation on a number of areas has enhanced and solidified our earlier work. We have formalized the $\mathcal{ZVM}$ to provide the basis for describing the mapping and integration of participating DBMSs, *Zeus* views, and applications. An IDL-based specification of the $\mathcal{ZMS}$ that details

the functionality of the $\mathcal{ZMS}$ server and clients has been completed. We have also investigated the issues involved in designing an object query language (OQL) for object database management systems (ODBMS). We now have a comprehensive survey and analysis of object query languages which have drawn the skeleton of a methodology for designing an object query language as an interface to the $\mathcal{ZMS}$.

Project *Zeus* has exciting prospects. CORBA IDL and Larch/CORBA will provide a complete specification tool for documenting our system independent of programming languages. The framework-based design method will be integrated with a metrics approach to provide an automatable way of filtering out favorable design alternatives. This will prove that our design method has the potential of simplifying the design of large-scale distributed systems.

BIBLIOGRAPHY

[1] Abiteboul, S., and A. Bonner. "Objects and Views," *ACM SIGMOD*, 1991, pp. 238-247.

[2] Agrawal, R., and N. Gehani. "ODE (Object Database and Environment): the Language and the Data Model," *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 365–386.

[3] Ahmed, A., A. Wong, D. Sriram, and R. Logcher. "Object-oriented database management systems for engineering: A comparison," *Journal of Object-Oriented Programming,* Jun. 1992, pp. 27–44.

[4] Ahmed, R., P. Smedt, and W. Du. "The Pegasus Heterogeneous Multidatabase System," *IEEE Computer*, Dec. 1991, pp. 19-27.

[5] Alashqur, A., S. Su, and H. Lam. "OQL: A Query Language for Manipulating Object-Oriented Databases," *VLDB*, 1989, pp. 433–442.

[6] Andrew, T., C. Harris, and K. Sinkel. "ONTOS: A Persistent Database for C++," *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 387–406.

[7] Bancilhon, F. "Object Database Systems: Functional Architecture," *Lecture Notes in Computer Science,* Vol. 742, First JSSST Int'l Symposium, Nov. 1993, pp. 163–175.

[8] Bancilhon, F., C. Delobel, and P. Kanellakis, Ed. *Building an Object-Oriented Database System: The Story of* O_2. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[9] Bancilhon, F., S. Cluet, and C. Delobel. "A Query Language for the O_2 Object-Oriented Database System," *Proc. of 2nd Database Programming Language Workshop,* 1990, pp. 122–138.

[10] Banerjee, J., W. Kim, and K. Kim. "Queries in Object-Oriented Databases," *Proc. 4th Int'l Conference on Data Engineering*, Feb. 1988.

[11] Barsalou, T., and D. Gangopadhyay. "M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems," *IEEE Data Engineering,* 1992, pp. 218-227.

[12] Barsalou, T., A. M. Keller, N. Siambela and G. Wiederhold. "Updating Relational Databases through Object-Based Views," *ACM SIGMOD*, 1991, pp. 248-257.

[13] Beech, D. "A Foundation for Evolution from Relational to Object Databases," *Advances in Database Technology - EDTB*, Vol. 33, 1988, pp. 251–270.

[14] Bernstein, P. A. "Middleware: An Architecture for Distributed System Services," Technical Report CRL 93/6, Cambridge Research Lab, Digital Equipment Co., Mar. 1993.

[15] Bertino, E., M. Negri, G. Pelagatti, and L. Sbattella. "Object-Oriented Query Languages: The Notion and the Issues," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 4, No. 3, Jun. 1992, pp. 223–237.

[16] Blakeley, J., C. Thompson, and A. Alashqur. "Strawman Reference Model for Object Query Languages," *Computer Standards & Interfaces,* 13, 1991, pp. 185–199.

[17] Blakeley, J., W. McKenna, and G. Graefe. "Experiences Building the Open OODB Query Optimizer," *SIGMOD*, 1993, pp. 210–216.

[18] Blakeley, J. "ZQL[C++]: Extending the C++ Language with an Object Query Capability," *Database Challenges in the 1990's*, Won Kim (Ed.), ACM Press, Addison-Wesley, Menlo Park, CA, 1993.

[19] Bloom, T., and S. Zdonik. "Issues in the Design of Object-Oriented Database Programming Languages," *Object-Oriented Programming Systems, Languages and Applications,* OOPSLA'87, pp. 441–451.

[20] Bretl, R., et. al. "The GemStone Data Management System," *Object-Oriented Concepts, Databases, and Applications.* Kim, W., and F. Lochovsky, (Ed.) Frontier Series, ACM Press, Addison Wesley, Menlo Park, CA, 1989, pp. 283–308.

[21] Bright, M. W., and A. R. Hurson. "Multidatabase Systems: An Advanced Concept in Handling Distributed Data," *Advances in Computers.* Vol. 32, 1991, pp. 149-201.

[22] Brodie, M. "On the Development of Data Models," *On Conceptual Modeling,* Springer Verlag, New York City, 1984.

[23] Brodie, M., and S. Ceri. "On Intelligent and Cooperative Information Systems," *Int'l Journal of Intelligent and Cooperative Information Systems,* Sep. 1992, pp. 121–131.

[24] Bukhres O., A. Elmagarmid, and J. Mullen. "Object-Oriented Multidatabases: Systems and Research Overview," *Int'l Conference in Knowledge Management,* 1992, pp. 27-34.

[25] Bukhres, O., J. Chen, A. Elmagarmid, X. Liu, and J. Mullen. "InterBase: A Multidatabase Prototype System," *SIGMOD,* 1993, pp. 534-539.

[26] Carey, M., et al. "The Architecture of the EXODUS Extensible DBMS," *Proc. Int'l Workshop on Object-Oriented Database Systems,* Pacific Grove, Sep. 1986.

[27] Carey, M., and D. DeWitt. "An Overview of the EXODUS Project," *Database Engineering,* Jun. 1987, pp. 110–129.

[28] Carey, M., D. Dewitt, and S. Vandenberg. "A Data Model and Query Language for EXODUS," *SIGMOD,* 1988, pp. 413–423.

[29] Cattell, R., Ed. *The Object Database Standard: ODMG-93,* Morgan Kaufmann, San Mateo, California, 1993.

[30] Champine, G. A., D. E. Geer, Jr., and W. N. Ruh. "Project Athena as a Distributed Computer System," *IEEE Computer,* Sep. 1990, pp. 40-51.

[31] Chan, D., D. Harper, and P. Trinder. "Object-Oriented Query Languages: Data Model Issues, Survey, Comparison and Analysis," Department Research Report, DB-1992-1, Dept. of Computing Science, Univ. of Glasgow, Nov. 1992.

[32] Chen, Q., and M. Shan. "Abstract View Objects for Multiple OODB Integration," *Lecture Notes in Computer Science,* No. 742, 1993, pp. 237–250.

[33] Chen, I. A., and D. McLeod. "Derived Data Update in Semantic Databases," *Very Large Data Bases,* 1989, pp. 225-235.

[34] Chung, C. "Dataplex: An Access to Heterogeneous Distributed Databases," *Communications of the ACM*, Vol. 33, No. 1, Jan. 1990, pp. 70-80.

[35] Cluet, S. "RELOOP, an Algebra Based Query Language for an Object-Oriented Database System," *Data & Knowledge Engineering,* 5 (1990), pp. 333–352.

[36] Codd, E. "A Relational Model for Large Shared Data Banks," *Communications of ACM*, 13(6), 1970, pp. 377-387.

[37] Collet, C., M. Huhns, and W. Shen. "Resource Integration Using a Large Knowledge Base in Carnot," *IEEE Computer,* Dec. 1991, pp. 55–62.

[38] Czejdo, B., and M. C. Taylor. "Integration of Information Systems Using an Object-Oriented Approach," *The Computer Journal*, Vol. 35, No. 5, 1992, pp. 501-513.

[39] Dar, S., N. Gehani, and H. Jagadish. "CQL++: A SQL for the Ode Object-Oriented Database," *Proc. of Int'l Conf. on Extending Database Technology,* Vienna, Austria, Mar. 1992, pp. 111-121.

[40] Dayal, U. "Multibase - integrating heterogeneous distributed database systems," *Proc. of AFIPS*, 1981, pp. 487-499.

[41] Deen, S., R. Amin, and M. Taylor. "Data Integration in Distributed Databases," *IEEE Transactions on Software Eng.,* Vol. SE–13, No. 7, Jul. 1987, pp. 860–864.

[42] Fichman, R., and C. Kemerer. "Object-Oriented and Conventional Analysis and Design Methodologies," *IEEE Computer,* Oct. 1992, pp. 22–39.

[43] Finkelstein, R. "Breaking the Mold," *Database Programming & Design*, Feb. 1993, pp. 39–43.

[44] Fishman, D., et al. "Overview of the Iris DBMS," *Object-Oriented Concepts, Databases, and Applications.* Kim, W., and F. Lochovsky, (Ed.) Frontier Series, ACM Press, Addison Wesley, Menlo Park, CA, 1989, pp. 219–250.

[45] Fong, E., W. Kent, K. Moore and C. Thompson. *X3/SPARC/DBSSG/OODBTG Final Report,* American National Standards Institute (ANSI), Aug. 1991.

[46] Gallagher, L. "Object SQL: Language Extensions for Object Data Management," *Int'l Conference on Information and Knowledge Management,* 1992, pp. 17–26.

[47] Gardarin, G., and P. Valduriez. "ESQL2: An Object-Oriented SQL with F-Logic Semantics," *IEEE Data Engineering,* 1992, pp. 320–327.

[48] Garlan, D. "Extending IDL to Support Concurrent Views," *ACM SIGPLAN,* Not. 22, 11, 1986, pp. 95-110.

[49] Geppert, A., et al. "The NO^2 Data Model," Technical Report 93.09, Computer Science Department, University of Zurich, Switzerland, Apr. 1993.

[50] Gien, M. "Next Generation Operating Systems Architecture," *Lecture Notes in Computer Science,* No. 563, 1991.

[51] Gupta, A., ed. "Integration of Information Systems: Bridging Heterogeneous Databases," *IEEE Press,* New York City, 1989.

[52] Harris, C., and J. Duhl. "Object SQL," *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 199–215.

[53] Heiler, S., and S. Zdonik. "Object Views: Extending the Vision," *IEEE Data Engineering,* 1990, pp. 86-93.

[54] Heimbigner, D., and D. McLeod. "A Federated Architecture for Information Management," *ACM Trans. on Office Information Systems,* Jul. 1985, Vol. 3, No. 3.

[55] Hornick, M. F., J. D. Morrison and F. Nayeri. "Integrating Heterogeneous, Autonomous, Distributed Applications Using the DOM Prototype," TR-0174-12-91-165, GTE Laboratories Incorporated, Waltham, MA, Dec. 1991.

[56] Horowitz, E., and Q. Wan. "An Overview of Existing Object-Oriented Database Systems," *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 101–116.

[57] Hurson, A., M. Bright, and S. Pakzad. *Multidatabase Systems: An Advanced Solution for Global Information Sharing.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[58] Guttag, J., J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification,* Springer-Verlag, New York City, 1993.

[59] Jarke, M., Ed. *ConceptBase V3.2 User Manual.* Aachen, 1993.

[60] Kaul, M., K. Drosten and E. J. Neuhold. "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views," *IEEE Data Engineering*, 1990, pp. 2-10.

[61] Keller, A., J. Richard, and S. Agarwal. "Persistence Software: Bridging Object-Oriented Programming and Relational Databases," *SIGMOD*, 1993, pp. 523–528.

[62] Kent, W. "Important Features of Iris OSQL," *Computer Standards & Interfaces*, 13(1991), pp. 201–206.

[63] Kifer, M., G. Lausen, and J. Wu. "Logical Foundations of Object-Oriented and Frame-Based Languages," Technical Report 90/14, Dept. of Computer Science, SUNY at Stony Brook, Aug. 1990.

[64] Kifer, M., W. Kim, and Y. Sagiv. "Querying Object-Oriented Databases," *SIGMOD*, 1992, pp. 393–402.

[65] Kim, W. "Features of the ORION Object-Oriented Database," *Object-Oriented Concepts, Databases, and Applications.* Kim, W., and F. Lochovsky, (Ed.) Frontier Series, ACM Press, Addison Wesley, Menlo Park, CA, 1989, pp. 251–282.

[66] Kim, W. "A Model of Queries for Object-Oriented Databases," *VLDB*, 1989, pp. 423–432.

[67] Kim, W. *Introduction to Object-Oriented Databases,* Computer Systems Series, the MIT Press, Cambridge, MA.

[68] Kong, M. *Network Computing System Reference Manual*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[69] Konstantas, D. "Object-Oriented Interoperability," *Lecture Notes in Computer Science,* No. 707, ECOOP, 1993, pp. 80–102.

[70] Lamb, C., G. Landis, J. Orenstein, and D. Sakahara. "The ObjectStore Database System," *Communications of the ACM*, Oct. 1991, pp. 50-63.

[71] Little, T., and A. Ghafoor. "Spatio-Temporal Composition of Distributed Multimedia Objects for Value-Added Networks," *IEEE Computer*, Oct. 1991, pp. 42–50.

[72] Loomis, M. "Object Database Semantics," *Journal of Object-Oriented Programming,* Jul.-Aug. 1993, pp. 26-33.

[73] Maier, D. "Development of an Object-Oriented DBMS," *OOPSLA*, 1986, pp. 472–482.

[74] Manola, F. "An Evaluation of Object-Oriented DBMS Development," TR-0066-10-89-165, GTE Laboratories Inc., Waltham, MA, 1989.

[75] Manola, F. "Object Data Language Facilities for Multimedia Data Types," TR-0169-12-91-165, GTE Laboratories Inc., Waltham, MA, 1991.

[76] McCarthy, J. "Metadata Management for Large Statistical Database," *Very Large Data Bases*, 1982, pp. 470–502.

[77] Mitchell, G., S. Zdonic, and U. Dayal. "An Architecture for Query Processing in Persistent Object Stores," CS-91-38, Dept. of Computer Science, Brown University, June 1991.

[78] Mitchell, G. "Extensible Query Processing in an Object-Oriented Database," CS-93-16, Dept. of Computer Science, Brown University, May 1993.

[79] Monk, S. "The CLOSQL Query Language," Technical report No. SE-91-15, Computing Dept., Lancaster University, Lancaster, UK. 1991.

[80] Mullen, G. "Supporting Queries in the O-Raid Object-Oriented Database System," *Proc. of the Int'l Computer Software and Applications Conference,* Oct. 1990, pp. 245–250.

[81] Nicol, J., C. Wilkes, and F. Manola. "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Jun. 1993, pp. 57–67.

[82] Nierstrasz, O., S. Gibbs and D. Tsichritzis. "Component Oriented Software Development," *Communications of the ACM*, Sep. 1992, pp. 160-165.

[83] Orenstein, J., S. Haradhvala, B. Margulies, and D. Sakahara. "Query Processing in the ObjectStore Database System," *ACM SIGMOD*, 1992, pp. 403-412.

[84] Ozsu, M., and D. Straube. "Issues in Query Model Design in Object-Oriented Database Systems," *Computer Standards & Interfaces,* 13, 1991, pp. 157–167.

[85] Peters, R. J., M. T. Ozsu and D. Szafron. "TIGUKAT: An Object Model for Query and View Support in Object Database Systems," TR 92-14, U. of Alberta, Oct. 1992.

[86] Peters, R., A. Lipka, M. Ozsu, and D. Szafron. "The Query Model and Query Language of Tigukat," TR 93-01, Dept. of Computing Science, Univ. of Alberta, Jan. 1993.

[87] Premerlani, W. J., M. R. Blaha, J. E. Rumbaugh and T. A. Varwig. "An Object-Oriented Relational Database," *Communications of the ACM*, pages 99-109, Nov. 1990.

[88] Pu, C. "Superdatabases for Composition of Heterogeneous Databases," *IEEE Data Engineering*, 1988, pp. 548-555.

[89] Rigney, T. "New SQL Database Supports Multimedia," *Communications Week*, Nov. 22, 1993, pp. 14.

[90] Roberts, J., and L. Miller. "A Prototype of the Generation of Views for an Object-Relational Interface," *ISCA*, 1993, pp. 138–141.

[91] Rosenberry, W., D. Kenney and G. Fisher. *Understanding DCE*, O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[92] Saltor, F., and M. Garcia-Solaco. "Suitability of Data Models as Canonical Models for Federated Databases," *SIGMOD RECORD*, Special Issue, Vol. 20, No. 4, Dec. 1991, pp. 44–48.

[93] Sarkar, M., and S. Reiss. "A Data Model and a Query Language for Object-Oriented Databases," CS-92-57, Dept. of Computer Science, Brown University, Dec. 1992.

[94] Scholl, M. H., C. Laasch and M. Tresch. "Updatable Views in Object-Oriented Databases," *Deductive and Object-Oriented Databases (DOOD)*, 1991, pp. 189-207.

[95] Shaw, G., and S. Zdonik. "A Query Algebra for Object-Oriented Databases," CS-89-19, Dept. of Computer Science, Brown University, March 1989.

[96] Sheth, A. "Semantic Issues in Multidatabase Systems," *SIGMOD RECORD*, Special Issue, Vol. 20, No. 4, Dec. 1991.

[97] Sheth, A., and , L. Kalinichenko. "Information Modeling in Multidatabase Systems: Beyond Data Modeling," *Int'l Conference in Knowledge Management*, 1992, pp. 8–16.

[98] Sheth, A. "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Survey*, 1990, pp. 79–104.

[99] Shilling, J. J., and P. F. Sweeney. "Three Steps to Views: Extending the Object-Oriented Paradigm," *OOPSLA*, 1989, pp. 353-361.

[100] Silberschantz, A., M. Stonebraker, and J. Ullman. "Database Systems: Achievements and Opportunities," *Communications of the ACM,* Vol. 34, No. 10, Oct. 1991, pp. 110-120.

[101] Stonebraker, M. "The Design and Implementation of INGRES," *ACM Trans. on Database Systems,* Sep. 1976, pp. 210–228.

[102] Stonebraker, M., and G. Kemnitz. "The POSTGRES Next-Generation Database Management System," *Communications of the ACM*, 1991, Vol. 34, No. 10, pp. 78–92.

[103] Straube, D. "Queries and Query Processing in Object-Oriented Database Systems," Ph.D. Thesis, TR 90-33, Dept. of Computing, Univ. of Alberta, Dec. 1990.

[104] Sudha, R. "Heterogeneous Distributed Database Systems," *IEEE Computer,* Dec. 1991, pp. 7-9.

[105] Su, S., S. Fang, and H. Lam. "An Object-Oriented Rule-Based Approach to Data Model and Schema Integration," *Technical Report (TR-93-12), U. of Florida,* 1993.

[106] Ty, Frederick. "G-OQL: Graphics Interface to the Object-Oriented Query Language OQL," Master Thesis, U. of Florida, 1988.

[107] Vandenberg, S. "Algebras for Object-Oriented Query Languages," TR 1161, Dept. of Computer Science, Univ. of Wisconsin, 1993.

[108] Veijalainen, J., and R. Popescu-Zeletin. "Multidatabase Systems in ISO/OSI Environment," *Standards in Information Technology and Industrial Control,* 1988, pp. 83-97.

[109] Weinreb D., et al. "An Object-Oriented Database System to Support an Integrated Programming Environment," *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 117–129.

[110] Weiser, S., and F. Lochovsky. "OZ+: An Object-Oriented Database System," *Object-Oriented Concepts, Databases, and Applications.* Kim, W., and F. Lochovsky, (Ed.) Frontier Series, ACM Press, Addison Wesley, Menlo Park, CA, 1989, pp. 309–340.

[111] Wells, D. L., J. A. Blakeley and C. W. Thompson. "Architecture of an Open Object-Oriented Database Management System," *IEEE Computer*, Oct. 1992, pp. 74-82.

[112] Wiederhold, G., P. Wegner and S. Ceri. "Toward Megaprogramming," *Communications of the ACM*, Nov. 1992, pp. 89-99.

[113] Wiederhold, G. "Views, Objects, and Databases," *IEEE Computer*, Dec. 1986, pp. 37-44.

[114] Wiener, J., and Y. Ioannidis, "A Moose and a Fox Can Aid Scientists with Data Management Problems," TR 1182, Dept. of Computer Science, Univ. of Wisconsin, 1993.

[115] Wileden, J., A. Wolf, W. Rosenblatt, and P. Tarr. "Specification-Level Interoperability," *Communications of the ACM*, May 1991, pp. 73–87.

[116] Wirfs-Brock, R., and R. Johnson. "Surveying Current Research in Object-Oriented Design," *Communications of the ACM,* Sep. 1990, pp. 104–124.

[117] Yen, C., and L. Miller. "An Extensible View System for Multidatabase Integration and Interoperation," to appear in the *Journal of Integrated Computer-Aided Engineering*, Nov. 1994.

[118] Yen, C. H., and L. L. Miller. "The Design and Implementation of the Zeus View System," *Proc. of the 27th Hawaii Int'l Conference on System Sciences*, Vol. II, 1994, pp. 206–215.

[119] Zicari, R. "The SOL Object-Oriented Database Language," *Lecture Notes in Computer Science,* No. 593, May 1992, pp. 105–127.

[120] *Open Systems Handbook: A Guide to Building Open Systems,* Cambridge Research Laboratories, Digital Equipment Corporation, 1991.

[121] *Leveraging Object-Oriented Frameworks.* White Paper, Taligent Inc., San Jose, CA, 1993.

[122] *The Common Object Request Broker: Architecture and Specification,* Object Management Group, OMG Document 91.12.1, Austin, Texas, Dec. 1991.

[123] *OMG Architecture Guide Chapter 4: The OMG Object Model,* Object Management Group, Austin, Texas, May 1992.

[124] "Interoperability: Communications Week's White Papers on Managing the Enterprise," *Supplement to Communications Week*, Oct. 12, 1992.

[125] *Information Processing Systems, Open Systems Interconnection, Remote Database Access - Part 1: Generic Model, Service and Protocol,* ISO-IEC-9579-1-DIS, Jun. 1991.

[126] *Information Processing Systems, Open Systems Interconnection, Remote Database Access - Part 2: SQL Specialization,* ISO-IEC-9579-2-DIS, Jun. 1991.

[127] *UniSQL Multidatabase System.* Data Sheets, UniSQL, Austin, Texas, 1992.

[128] "A New Direction in DBMS," An interview with Dr. Michael Stonebraker, *DBMS Client/Server Computing,* Feb. 1994, pp. 50–60.

[129] *Object Management Architecture Guide*, Revision 1.0, OMG TC Document 90.9.1, Austin, Texas.

[130] *The SOM Toolkit User's Guide,* Release 2.0, IBM Corporation, Austin, Texas.

[131] *ANSA ORB Interoperability.* Object Management Group, OMG Document 93-5-9, Austin, Texas, May 1993.

[132] Special Issue on Next-Generation Database Management Systems, *Communications of the ACM*, 34(10), 1991.

[133] "ITASCA Distributed ODBMS," Itasca Systems Inc., Minneapolis, Minnesota, May 1993.

[134] *OpenODB from Hewlett-Packard.* Technical Data, Hewlett-Packard Company, Palo Alto, CA, 1992.

[135] "Object Technology at Hewlett-Packard," *Object-Oriented Strategies,* Vol 3, No. 4, Paul Harmon (Ed.), Cambridge, MA, 1993.

[136] "Object-Oriented Database Management System Products," *Object-Oriented Strategies,* Vol. 2, No. 2, Paul Harmon (Ed.), 1992.

[137] *ObjectStore Technical Overview.* Release 2.0, Object Design Inc., Burlington, MA, Jul. 1992.

[138] *ObjectStore Product Profile.* Object Design Inc., Burlington, MA, 1993.

[139] "ONTOS Object Database and Tools for Rapid Object Application Development," ONTOS Inc., San Jose, CA, 1993.

[140] *Objectivity: Data Sheet.* Objectivity Inc., Menlo Park, CA, 1993.

[141] *O_2 Technology: Product Profile.* O_2 Technology, Paris, France, 1993.

[142] "GEMSTONE: The High-Performance Object Database Management System," Servio Corporation, San Jose, CA, 1993.

[143] *UniSQL News Release.* UniSQL Inc., Austin, Texas, 1993.

[144] *Versant Object SQL.* Versant Object Technology, Menlo Park, CA, 1993.

[145] *Object-oriented databases with applications to CASE, networks, and VLSI CAD.* R. Gupta and E. Horowitz, Ed., Englwood Cliffs, NJ : Prentice Hall, 1991, pp. 365–386.

[146] ISO/IEC SQL Revision. *ISO-ANSI Working Draft Database Language SQL (SQL3)*, Jim Melton, Ed., Document ISO/IEC JTC1/SC21 N6931, American National Standards Institute, New York, July 1992.

APPENDIX A. OBJECT QUERY LANGUAGES

A.1 Introduction

The advances in object database management systems (ODBMSs) have picked up speed in recent years due to the intensive research[132], the emerging products and prototypes[74, 141], and the large number of successful experiences with the applications built on ODBMSs[3, 145]. To compete against the relational database management systems (RDBMSs), ODBMSs must enhance the query language support which is one of the major strengths of RDBMSs. Recent advances in the development of ODBMSs have started to address the importance of query languages[29]. Major ODBMS vendors are striving to implement object query languages (OQLs) that not only aim at the essential needs of the query facility, but also conform to standards for interoperability. Despite these advances and the fact that the standardization of OQLs is under way[29, 46], lack of an extensive characterization and a taxonomy of OQLs has clouded the role and meaning of an OQL in an ODBMS. This has weakened the acceptance of OQLs in the user community. Eventually, the advances of both OQLs and ODBMSs may be slowed down due to the confusion.

In this Appendix, we summarize our survey and analysis of existing OQLs by characterizing their features. For the remainder of this section, we describe the motivation of this survey. Section A.2 looks at the basics of OQLs. Section A.3

describes how we characterize OQLs. The survey and analysis of existing OQLs are provided in Sections A.4 and A.5. Section A.6 covers the current standardization activities of OQLs.

Motivation. Most ODBMSs have interfaces built for one or more programming languages. As opposed to the declarativeness of query languages, programming languages are computationally-complete. However, programming languages lack the simplicity of query languages. Therefore, there is a need for a way of manipulating data from ODBMSs without having to write a program. An object query language or an ad hoc query facility are designated for this kind of functionality which indicates the irreplaceable role of an OQL in an ODBMS. We want to clearly identify what an OQL is, what it is used for, how it is used, how it should be designed, and so on. To answer all these questions, we must understand the characteristics of OQLs in general and the features of existing OQLs in particular. The nature of OQLs is complicated by a multitude of existing approaches to OQLs as well as the complexity of the hosting ODBMs. Future advances and developments of OQLs would have guidance toward success with a taxonomy for OQLs. To create such a taxonomy, we start with the characterization of an object query language along four dimensions: foundation, functionality, language environment, and language features. Within these dimensions, we explore the characteristics and issues of OQLs independent of existing OQLs. This leads to a common framework for characterizing existing OQLs. After surveying and analyzing existing OQLs, this common characterization framework can be refined to a taxonomy for OQLs.

A.2 Preliminaries

The functionality of a database language is to provide features for defining, retrieving and manipulating data. In an ODBMS, a database language often appears as an extended programming language plus some querying constructs. For example, an object-oriented programming language with persistence or an embedded object query language can be used as a database language. The querying constructs may be provided as a query language or simply as a programming interface. As far as the user interface is concerned, there are a number of alternatives. For example, the graphical user interface or graphical query-by-example. However, a query language is thought by many to be an essential feature of an ODBMS. In this survey, we focus on the query language part of an ODBMS. For the remainder of this section, we briefly overview what an OQL is, what OQL design issues are, and some of the terminology commonly used in OQLs and ODBMSs.

A.2.1 Semantics of Object Databases & Object Query Languages

To understand what an object query language is, we must know its meaning; i.e., semantics, and that of its hosting ODBMS. Object database semantics have been addressed in [72]. The semantics of an object database and an object query language are based on an object model. The basic semantic elements of an object model include the object, the type hierarchy that describes the relationships of objects, and the state and behavior of objects. An application's abstract object model is developed based on these semantic elements of the underlying object database. The implementation of such an abstract object model may be bound to a specific programming language and/or a query facility.

What an OQL is. An OQL provides an implementation-independent way of expressing queries against an ODBMS. The query may be intended to retrieve or manipulate data stored in an ODBMS. The syntax and semantics of an OQL fully specify how the queries should be handled, or how the result should be presented. An OQL is also independent of applications although the expressiveness of the OQL must accommodate the requirements of applications. A number of OQL semantic issues remain unsettled. This is why existing OQLs have several fundamental differences. For example, a collection of objects may be queried by users. Should these objects have the same type? If we allow a query to inquire a hierarchy of classes, what would be the type extents? Should users be allowed to explicitly formulate the type extent? Another example is the representation of objects in OQLs. There are two common approaches: value-based, and reference-based. Both approaches have implications on the use, design, and implementation of OQLs. Finally, should we allow a query to create objects that have new types?

Why we need an OQL. An OQL is required in an ODBMS for a number of reasons. First of all, an OQL allows queries to be composed declaratively. Queries may be used for associative search which is important in many application domains. Secondly, the declarativeness of queries makes the optimization easier. Thirdly, an OQL may allow an ODBMS to be used in both business and engineering applications. Finally, a programming language independent OQL is easier to standardize, and to serve as a DBMS interoperability tool.

OQL vs. Programming Language. OQLs and programming languages may have different object models. The functionality of OQLs and programming

languages complement each other nicely since there are different uses that require different ways to access the ODBMS. Some of the uses require features that are more easily incorporated in an object query language; e.g., associative search. Some of the uses may require features that are more appropriately coded in a programming language; e.g. complex procedures. The approach of extending the programming language to provide query facilities has its own limitations. Such an approach does not allow OQLs and programming languages to co-exist in ODBMSs and play different roles. There are a number of salient features of OQLs that separate them from programming languages:

1. The syntax of an object query language is declarative and easier to use in terms of getting the requested data. The semantics of an object query language can be derived from the underlying object database semantics. A programming language requires extensions to adjust to the object database semantics. Using a program to access an ODBMS requires the access details to be described in the program.

2. An object query language can describe abstract semantics independent of implementations. This implies multi-lingual support, application portability and interoperability.

3. Many existing object query languages have been integrated with programming languages. A declarative object query language may converge with programming languages by including specifications of operations. For the users of ODBMSs, this means a flexible and high-level combination of system and user interface facilities.

A.2.2 Design Issues

Based on the OQL semantics we discussed in the previous section, we can now explore the issues of designing such a query facility. We describe the OQL design issues along four dimensions: foundation, functionality, language environment, and language features.

Foundation. The foundation of an OQL includes an object model and a query model. It is common for an ODBMS to have an object model. Existing OQLs may or may not rely on the object model of the hosting ODBMS. Having the same object model for both the OQL and the ODBMS simplify the design of the OQL. However, in this case, the object model must be capable of supporting both the OQL and the ODBMS. A formal query model provides the basis for query optimization although most existing OQLs were not built upon a formal query model. With a formal query model, the formal semantics of an OQL can easily be defined.

Functionality. The functionality of an OQL depends on who uses it, what it is used for, and how it is used. Unlike an application program interface, a query facility should be easy to use. Most OQLs provide declarative and associative access to ODBMSs. Some existing OQLs are embedded in or integrated with programming languages to support complex queries that may involve computations. A casual user may choose to use an ad hoc query language to compose retrieval-oriented queries. An application programmer may prefer to use an OQL with a programming language. The query result and how it is presented are also important issues related to the functionality of an OQL.

Language Environment. The language environment describes how an OQL is provided, how it interacts or is integrated with other ODBMS components. An OQL may be separated from the object definition language (ODL). There may even be another language for describing the behavior of methods. Both the ODL and OQL can be made programming language independent. There might exist different degrees of integration and interactions between OQLs and programming languages. If the OQL is closely integrated with a programming language, some ODBMS-specific issues may be propagated from the OQL to the programming language; e.g., object persistence.

Language Features. Language features are important in the design of an OQL. Language features determine the characteristics, syntax, and semantics of an OQL. There are some high-level characteristics that should be decided; e.g., whether an OQL is computationally complete or not, or whether an OQL should preserve encapsulation, and so on. An OQL may have an abstract syntax which can then be tied to a concrete syntax for merging the query language into a specific programming language.

A.2.3 Terminology

Figure A.1 lists the acronyms commonly used in the discussions of OQLs and ODBMSs. In this section, we select a number of terms related to OQLs and provide detailed explanations. The terms are grouped in two categories: object data management and formal query models. A detailed object data management glossary can be found in [45]. Discussions of formal query models in ODBMSs can be found in

API	Application Program Interface
DBSSG	X3/SPARC Database Systems Study Group
DDL	Data Definition Language
DML	Data Manipulation Language
ODM	Object Data Management
ODMG	Object Database Management Group
ODBMS	Object Database Management System
OODBS	Object-Oriented Database System
OMG	Object Management Group
OODBTG	X3/SPARC/DBSSG Object-Oriented Database Task Group
OQL	Object Query Language
ORB	Object Request Broker
SPARC	X3 Standards Planning and Requirements Committee
SQL	Structured Query Language
SQL3	ISO-ANSI Working Draft Database Language SQL (SQL3)

Figure A.1: Acronyms.

[17, 66, 67, 77, 78, 84, 95, 103, 107].

A.2.3.1 Object Data Management In [45], the scope of the term, object data management, is identified to include object models and DBMSs. On the other hand, the term, object information management, broadens this scope to additionally include the use of objects in programming languages, network management, design methodologies, user interfaces, and related areas. We chose to use the term object rather than object-oriented since there exist object-based data management systems that are not object-oriented.

∇ **Abstract Looping.** A mechanism for examining the contents of aggregates without knowing the underlying data structures or implementations. For example, an

iterator.

∇ **Aggregate, Complex Object & Composite Object.** An aggregate is a collection of objects that may or may not have the same type. Therefore, an aggregate may be homogeneous or heterogeneous. An extensional aggregate refers to an aggregate whose identity is based on membership. An intentional aggregate refers to an aggregate whose identity is based on its creation event and its membership may change without affecting its identity. Complex objects are built from simpler ones by applying constructors to them; e.g., tuples, sets, arrays, bags, lists, and so on. Composite object is defined as an object with a hierarchy of exclusive component objects. Composite objects capture the IS-PART-OF relationship.

∇ **Application Program Interface.** A definition of the syntax and semantics that allows an application written in a host programming language to access or control an ODM system.

∇ **Associative Database Access.** Also refered to as value-based database access. As opposed to navigational (i.e., reference-based) access, associative access allows parallel access/retrieval (or content-addressed lookup) of data based on a predicate.

∇ **Attribute.** Also referred to as the property or instance variable of an object. All these terms refer to the object information that represents the visible part of the state of an object, and can be manipulated by **get** and **set** operations or other operations with similar semantics.

∇ **Computationally Complete.** A property of a programming language that includes control constructs for defining procedures or operations. A purely declarative language is not computationally complete.

∇ **Delegation.** The concept of an object assigning or transferring to another object the responsibility for defining an object, performing an operation, or implementing the state. Also called instance-instance inheritance.

∇ **Embedded Language.** A language whose commands are placed directly in another host programming language.

∇ **Generalization & Specialization.** Generalization refers to the action or process of deriving from many objects a concept or principle that is applicable to all the objects. For example, a base class is a generalization of its derived classes. Specialization refers to the action or process of adding a concept, principle, or operation to a class or type that is more specific or particular than other similar classes or types.

∇ **Impedance Mismatch.** The presence of incompatibilities between the data model of a programming language and that of a database. An ODM system has the property of seamlessness if the impedance mismatch is resolved in the sense that the data models of the database and the host programming language are the same syntactically or semantically, the host DML can be used as the ODM DML, or the ODM API is a natural extension to the host programming language.

∇ **Integrity Constraint.** A predicate that states a condition that must hold for a system or an object to be in a legal state or that defines legal state transitions. Referential integrity is a kind of integrity constraint that guarantees that all referenced objects exist.

∇ **Meta Data.** Meta data refer to the information about data. In an DBMS, meta data refer to database schemas that describe the data stored in the database. In an object model, meta data refers to type definitions or class definitions that describe the structure and behavior of their instances.

∇ **Persistence.** A property of an object or a programming language. A persistent object exists longer than the process that created it. A persistent programming language is a language in which the values of some or all data elements survive an execution of a program written in the language.

∇ **Reference Model.** A model that provides a framework to understand the key ideas and scope of a paradigm, to define the common characteristics of a system, and to provide a basis for comparing similar systems.

∇ **Reflective/Reflexive Systems.** Reflexive systems often refer to systems implemented in terms of themselves. Reflective systems are systems with Meta-Object Protocol; i.e., meta-classes are accessible to users. Reflective systems are inevitably reflexive.

∇ **Transitive Closure.** A property of an operation that either returns all objects reachable from an object, or returns all objects reachable from an object subject to a boundary condition.

A.2.3.2 Formal Query Model A query model describes how a query is expressed, processed, and optimized. A formal query model normally consists of a calculus, an algebra, and a complete query processing methodology. The research in the query models for ODBMSs has yet to explore many relevant issues. Although most terms have similar meanings in query models for ODBMSs and RDBMSs, some of the terms do differ due to the differences between ODBMSs and RDBMSs.

∇ **Calculus & Algebra.** Ozsu and Straube gave a definition of calculus and algebra in [84]: "A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in an algebra are procedural in nature but can be optimized." The query language must be equivalent to the calculus in expressive power. The calculus representations can be translated to algebra representations which can be optimized and used to generate the execution plan for accessing the DBMS. An algebra can be used in different ways. For example, an algebra can be used as the formal semantics and the theoretical foundation of a query language. An algebra can also be used as a framework for query optimization. In some cases, an algebra can be directly used as a query language.

∇ **Completeness, Safety & Closure.** Completeness of a query model requires

the calculus and the algebra to be equivalent in expressive power. Safety of calculus expressions guarantees that queries retrieve a finite set of objects in a finite amount of time. The closure property of an object algebra requires that the operators of the algebra operate on objects of a type in the type system and returns a set of objects whose types exist in the type system.

A.3 Characteristics

We characterize the OQLs along four dimensions: foundation, functionality, language environment, and language features. The foundation provides the basis for describing the modeling power, expressiveness, and features of OQLs. The functionality describes what an OQL is used for; i.e., the application requirements that are provided by the OQL. To develop an OQL, we need to know the language environment in addition to its foundation and functionality; e.g., how an OQL interacts with other components of the hosting ODBMS. Finally, the features of an OQL are determined in such a way that the application requirements are met and the development is feasible under the constraints of the language environment.

A.3.1 Foundation

The foundation of an OQL is composed of an object model and a query model. Whether an OQL should have the same object model as that of its hosting ODBMS remains a design decision. However, for the sake of the discussion we assume that an OQL does have the same object model as that of its hosting ODBMS. We use the object model described in [29, 123] as a reference object model. The query model for ODBMSs has recently been investigated[17, 66, 67, 77, 78, 84, 95, 103, 107].

A.3.1.1 Object Model The ODMG object model is deemed as a superset of the OMG object model[29]. We use the ODMG object model as our reference model. In this section, we describe both the OMG object model[123] and the ODMG object model, and explain how an object model may affect an OQL.

A Reference Object Model. A reference object model (ROM) is used to provide a common basis for modeling real-world and conceptual entities. The ROM is composed of a core object model and a set of components which are compatible extensions to the core object model. The core object model along with different combinations of the components provide the modeling facility for a particular application domain. The ROM is extensible in the sense that components can be added to tailor to the requirements of existing and future application domains. The basic modeling construct is an object. A type has a unique identifier, an interface, and an extension. An object is an instance of a type. The state and behavior of an object are the value and implementations of the properties inherited from the interface of the object's type. The extension of a type is the set of its instances. A class is a collection of objects that have the same type and the same implementation of the type's interface. We define a type system as a group of types that model a specific application domain. The set of instances of the types in a type system forms an object system. The object model of ODMG-93 provides as components a number of compatible extensions to the OMG core object model. These extensions along with the core object model create a object model for ODBMSs. The extensions include the notions of attributes and relationships, structured objects, built-in type hierarchies, and transactions.

Object Model and OQL. A conceptual data model is the foundation for describing a query language. Similarly, an object model determines the semantics of an OQL. An object model is the basis for describing the structure, state, and semantics of real-world entities which are required by an OQL to provide different views of the object database for users.

A.3.1.2 Query Model A query model provides a formal basis for describing the syntax, semantics, and processing methodology of queries. The query facility of the RDBMSs is based on a formal query model[36]. The lack of consensus on the formalism of existing object models has hampered the progress of formal query models for ODBMSs. The relevant issues on query models in ODBMSs have not been fully investigated[84]. It is hard to say whether a query model is essential for the success of ODBMSs. However, a formal query model will certainly help the design of OQLs and future standardization efforts. In general, a query model consists of a formal description of the syntax and semantics of queries as well as a complete query processing methodology. The syntax and semantics of queries may be a calculus, an algebra, and a set of algorithms for the transformation between the two. The query processing methodology includes the interpretation, optimization, and execution of queries. Query model issues of ODBMSs have been extensively addressed in [84].

A.3.2 Functionality

The functionality of an OQL addresses what an OQL accomplishes and how. The basic functionality of an OQL may include the definition, creation, update, retrieval, and deletion of the object and type information. In the actual design and

development of ODBMSs, the above-mentioned basic functionality may spread over a number of languages; e.g., DDL, DML, OQL, etc. The way an OQL achieves the basic functionality spawns detailed requirements. For example, *in the context of defining objects and types*, an OQL may allow a composite object to be defined over a set of objects of different types. Then, the created composite object may have a new type. Or, in some cases, we may want to avoid creating a new type for a composite object in order to keep the type hierarchy from growing. *In the context of creating and deleting objects/types*, an OQL may provide constructors and destructors similar to C++. These constructors and destructors may be implicitly or explicitly invoked upon creation or deletion of objects/types. *The retrieval or querying of objects/types involve many interesting features of an OQL.* For example, what can be queried? What type of object/data can be returned from a query? Can new objects and/or new types be created via queries? If new types can be created from queries, where should these new types be placed in the existing type hierarchy? Can the returned result be further queried? Are value joins and/or entity (reference) joins allowed? Can the join be done across object collections? How is the scope of the query defined? Can a query be used against multiple collections? If so, how will the query result be presented and used? Is the query always against a single class, a sub-graph or the entire graph? What kind of types and type extents can be queried? Can both transient and persistent objects be queried?

A.3.3 Language Environment

The language environment of an OQL describes how an OQL interacts with other components of the hosting ODBMS. This kind of information is not captured by the

syntax or semantics of the OQL. However, the language environment of an OQL does provide information regarding the design, implementation and performance of the OQL. For example, an OQL of an ODBMS may have several components like DDL and DML. It is also possible that the query facility is provided as a number of other components of the hosting ODBMS in addition to the OQL. In both cases, the hosting ODBMS must support an integrated environment such that users have a uniform querying interface. The low-level save/restore scheme is not an issue in most OQLs. Although it may affect the use of OQLs when both persistent and transient objects are allowed to be queried. The same issue is true as to what kind of persistence is provided. Most OQLs are declarative and are often integrated with a programming language. Different degrees and types of integration are possible.

A.3.4 Language Features

Language features describe the syntax and semantics of the OQL. How the functionality of the OQL is provided and how users query and view the object database are determined by the language features of an OQL. Knowing the language features of an OQL can provide insight into how the OQL issues are resolved or supported; e.g., side effects of methods and database integrity.

A.3.4.1 Object-Orientation Common object-oriented properties like encapsulation, inheritance, and polymorphism are normally supported in an OQL, although there might be differences in different existing OQLs. Different degrees of encapsulation can be supported in an OQL. Full encapsulation will ensure that the access to object properties is always through externally visible methods. Partial or

no encapsulation allows direct access of object properties. There also exist different types of inheritance; e.g., partial inheritance, single inheritance, and multiple inheritance. Polymorphism can be supported in the form of overloading, coercion, inclusion, or generalizing.

A.3.4.2 Type Definition & Object Management Queries are used over collections of objects of the same or different types. The definition of types creates a type hierarchy that captures the relationships between types. The instances of types can be grouped in different ways to form different type extents which correspond to different partitions of the object database. If both persistent and transient objects are allowed to be queried, both persistent and transient types should be allowed to be defined in the OQL. Object management supports the creation, deletion, save/restore, and update of objects in the object database.

A.3.4.3 Expressive Power The expressive power affects the use of an OQL. An OQL should be simple and easy to use while maintaining the capability of expressing the precise semantics of both simple and complex queries. Although the syntax may differ, there are many common features of existing OQLs that enhance the expressiveness of queries through language constructs and extended semantics.

A.3.4.4 Other Supporting Features Some language features are useful although they are not always necessary in most queries. For example, built-in aggregate functions provide pre-defined functions or operators that can be applied to aggregates. Such built-in aggregate functions can often be used to compute the sum, average, etc., over a property of a set of objects. Import/export facilities are also

examples of other supporting features of OQLs. Import/export facilities can help exchange data between the ODBMS and other software systems.

Existing OQLs will be surveyed and analyzed in the next two sections. We group existing OQLs into two categories based on their hosting ODBMSs: OQLs of commercial ODBMSs, and OQLs of the ODBMS resulting from proposals and research prototypes. The presentation of our survey is laid on the framework described in Section A.3. The analysis is focused on the distinguishable features of the individual OQL.

A.4 Survey of Commercial ODBMSs

A.4.1 Servio

GemStone is an OODBMS developed by Servio Corporation[20, 73, 142]. Features of GemStone include active database, concurrent support for multiple programming languages, multi-user transaction control, object-level security, dynamic schema and object evolution, legacy gateways, etc. Associative access can be done through path expressions and instance variable typing in OPAL. The basic components of the GemStone architecture are the Gem server process and the Stone monitor. The Gem server is where object behavior specified in GemStone's DML is executed. Each Gem server autonomously performs all actions necessary to commit a transaction. The Stone monitor coordinates such activity by multiple Gems.

In GemStone, an entity is modeled as an object. Properties of entities can be simple data values or other entities of arbitrary complexity. Set-valued objects are supported. Sets can have arbitrary objects as elements and need not be homogeneous. Gemstone supports object identity. To reduce *update anomalies* that exist in the

relational data model, entities with information in common can be modeled as two objects with a shared sub-object containing the common information. A library of classes implementing frequently used data types is provided. Classes are organized in a class hierarchy. Subclassing is supported to capture similarities among various classes of entities that are not totally identical in structure or behavior.

GemStone provides an object-oriented database language called OPAL which is used for data definition, data manipulation and general computation. OPAL is computationally complete. The query language of GemStone is provided as a limited calculus sub-language. Associative queries can be viewed as procedural OPAL code. Selection of collections is supported for subclasses of type **set** and **bag**. GemStone chose to index on collections instead of classes. For use with indices, the path syntax has been added to the OPAL language. For any variable, we can append to it a path composed of a sequence of links which specify some sub-part of an object. Selection conditions for associative access are conjunctions of comparisons. The comparisons are between path expressions and other path expressions or literals. OPAL provides typing for names and anonymous instance variables. Both named and anonymous instance variable typings are inherited through the type hierarchy.

A.4.2 HP

OpenODB is a commercial ODBMS from the Hewlett Packard Company. OpenODB has a client-server architecture. OpenODB clients include an interactive object-oriented SQL (IOSQL), a graphical browser, a programmatic interface, user applications and tools. An object-oriented SQL (HP OSQL) is provided as a query interface. IOSQL allows users to interactively enter IOSQL statements and query the object

database. The graphical browser allows users to graphically explore the database schema and contents. The programmatic interface is similar to a "*Dynamic SQL*" interface, and exchanges strings representing OSQL statements with the OpenODB server. User applications and tools developed using IOSQL, the graphical browser or the programmatic interface are all referred to as OpenODB clients. OpenODB server components include an object manager, a relational storage manager, and external functions. An object-oriented SQL (HP OSQL) is provided as an object-oriented front-end for relational databases. OSQL can be used for defining, creating, and manipulating objects, types, and functions. In OSQL, users only need to specify what data to retrieve instead of how to get the data. OSQL is a functional language that is a semantic superset of SQL. In addition to supporting a common query facility, OSQL is also used for application development.

The object model of OpenODB has three major components: objects, types, and functions. Objects are a combination of data and stored code that operate on the data. Types are used to classify similar objects. Functions operate on data in the database and also define the behavior of that data in the database. OpenODB supports three types of user-defined functions: stored functions, OSQL-based functions, and external functions. Stored functions define attributes and relationships that are stored in the database. OSQL-based functions define attributes and relationships that are retrieved or calculated with OSQL statements. External functions are a reference to code or data stored outside of OpenODB. Basic types and functions can be created using the CREATE TYPE statement. Additional functions can be created using the CREATE FUNCTION statement.

A.4.3 ITASCA

The ITASCA Distributed ODBMS was derived from the ORION series of object database research prototypes which were researched and developed at the Microelectronics and Computer Technology Corporation (MCC)[133]. The ITASCA product is an extension of the third and final ORION prototype from MCC. ITASCA is a fully distributed ODBMS. It has a client-server architecture. Both the processing and data can be distributed. Multiple servers and multiple clients are supported. The goal of the ITASCA design is to achieve transparent data access and to avoid single point of failure.

ITASCA uniformly models any real-world entity as an object. Each ITASCA object has a unique identifier along with state and behavior. Attributes represent the state of an object while methods define the behavior of the object. Classes and instances are both first-class objects. ITASCA supports attribute methods at both the instance level and the class level. Class objects may receive messages exactly like instance objects. Subclasses derive from existing classes. The resulting database schema forms a class lattice. Each subclass inherits all the attributes and methods defined for its superclasses. Multiple inheritance is supported. ITASCA supports composite objects built from component objects. Inverse links are maintained between parent and child objects. Child objects can be independent of their parent objects. Child objects can also be shared by several parent objects. If a child object is dependent upon its parent, it is deleted when its parent is deleted.

An ITASCA database contains both shared and private database partitions. The shared database contains objects that are accessible to all applications. Private databases contain objects that are localized to a given user or group of users. Appli-

cations can be written to access shared or private databases. ITASCA applications can be written in C++, CLOS, Lisp, C, and C callable languages. *ITASCA objects are represented in a neutral format independent of any programming languages.* Therefore, objects can be created using one programming language and then be updated or accessed using another programming language. The ITASCA query language is a 4GL interactive language which may be embedded in a program. ITASCA database methods are written in a Lisp-based 4GL which can reuse existing C, FORTRAN, or Lisp code. The query language supports text pattern matching of stored text as part of ITASCA's multimedia data management. ITASCA is an active database. Methods can be stored and activated directly in the database. This feature allows database methods to be changed without recompilation or relinking the application code. Also, the database methods or objects can be reused among multiple programming languages.

A.4.4 Objectivity

Objectivity/DB[140] is a commercial ODBMS developed by Objectivity Inc. Objectivity/DB has a distributed runtime architecture with a number of subsystems. The *Programming Interface* allows applications written in C or C++ to communicate with Objectivity/DB. The *Type Manager* stores, retrieves and maintains descriptions of all classes defined in the database. The *Object Manager* maintains and manipulates objects within the database. The *Storage Manager* is responsible for the physical placement of data in virtual memory and physical storage. The *Lock Server* coordinates access to all objects in the database and supports concurrency management among multiple users, multiple hosts, multiple databases, and multiple networks.

The *Network Manager* coordinates communication between processes, allowing local processes to transparently access data located on remote workstations. The *Operating System Interface* provides basic services for all other software in the system. Object query and traversal is provided as part of the development environment. A range of database facilities is supported; e.g., transaction management, recovery, etc.

In Objectivity/DB, an object is the fundamental storage entity that can be accessed and manipulated by Objectivity/DB applications. Objects are stored on pages. When an object is first referenced in a transaction, it is converted to the proper machine format. The Objectivity/DB object model follows the C++ object model. Object persistence is through inheritance from persistent classes. Objectivity/DB supports entity-relationship data models through the use of associations. An association is a logical link used to indicate that a relationship exists between two persistent objects. Associations can indicate 1:1, 1:N, and N:M relationships. *Objectivity/DB allows objects spanning multiple remote databases to participate in an association.* Associations are type safe, and are declared as part of an object class. Associations can be declared on both object classes involved in the relation. This kind of association is called bi-directional and allows the application to traverse between related objects through such links. This feature also helps maintain referential integrity in Objectivity/DB. Composite objects that group arbitrary set of objects of any classes are allowed. Objects can be linked via dynamic associations. In Objectivity/DB, types are defined as full objects.

Standard cfront C++ can be used as the DDL for Objectivity/DB schemas. The Objectivity/DB DDL is a high-level data modeling language that is a superset of the standard C++, with extensions for associations. Methods for using associations

are automatically generated by the Objectivity Schema Processor. Composite objects are modeled by specifying behavior attributes for Objectivity/DB associations. These attributes define what happens if an action should occur to an object participating in a relationship. Therefore, applications can model an arbitrary number of related objects as a single composite object. The Objectivity/DB variable sized array (varray) classes can be used to model complex objects. An object class may include any number of varray, and elements may be added or removed at run time. Objectivity/DB supports unordered collections. The methods provided at each level of the Objectivity/DB object containment hierarchy can be used to determine what objects are contained by another. *Objects in Objectivity/DB may be given a name within a name scope defined by the user.* Any object can be used to define a name scope, and an object may be given a name in several name scopes. Objectivity/DB supports traversal of objects in a traditional navigational (pointer chasing) fashion. Abstract looping is provided through an iterator which is a subclass of a handle. Objects may be accessed via handles and handles are type compatible with object references. The Objectivity/DB library interfaces provide full and consistent support for iterators. Objectivity/SQL++ provides associative query facility and allows queries to be formulated using a subset of SQL syntax. Standard C++ constructs can be used with query statements.

A.4.5 Object Design

ObjectStore[70, 83, 137, 138] is an ODBMS produced by Object Design, Inc. ObjectStore has a client-server architecture. The server process manages physical storage and arbitrates among client processes making requests for the data. The client

requests pages from the server in response to page faults generated by the application. The ObjectStore server only deals with pages. The client handles the query and the management of objects. The ObjectStore client libraries provide the interface between the user's application and the server. This interface manages the logical view of the data including collections, queries, versions, transaction management, memory management, and relationships among objects. *In other words, the bulk of the database functionality and the application logic reside on the client side.*

ObjectStore supports the complete C++ object model. Basic C++ types, virtual functions, inheritance, polymorphism, encapsulation, and parameterized types are all supported in ObjectStore. In addition, facilities are provided for modeling object collections, relationships between objects, and versioning of objects. In ObjectStore, collections are groupings of objects of the same type. Default collection behaviors are provided; e.g., insertion, removal, and retrieval of collection elements, set-theoretic operations such as union and intersection, and set-theoretic comparisons. Objects may have "embedded collections" that are arbitrarily large. In ObjectStore, queries can be expressed using an extended C++ supported by an extended C++ compiler, or through function calls belonging to a library interface.

In ObjectStore, queries are integrated with the host programming language in the form of query operators whose operands are a collection and a predicate. A library of collection classes like **sets**, **bags**, and **lists** are provided along with default collection behaviors. The collection facility is provided for both parameterized and non-parameterized classes. ObjectStore supports *orthogonal persistence.* Pointers to persistent objects are used in the same way as pointers to transient objects that have been dynamically allocated in the program's virtual address space. An ObjectStore

collection class can be either persistent or transient.

In ObjectStore, the structure of an object database are realized by inter-object references. Objects are located by traversing these references and by associative queries. Objects may be created, updated, or deleted. An ObjectStore query is an expression specified by a query operator. A query expression evaluates to a collection, a single object, or a boolean. Nested queries are supported. In a nested query, each query has its own range variable named **this**. In cases where both inner and outer range variables need to be referenced, an "alias" can be introduced for the outer range variable. Iteration is provided as a loop construct. The elements of a collection can be accessed one at a time via a **cursor** created as an instance of the class **os_Cursor**. For ordered collections, the default order of iteration is the order the elements take within the collection itself. For unordered collections such as sets and bags, the default iteration order is arbitrary.

A.4.6 Ontologic

ONTOS[139], formerly VBASE, is an ODBMS developed by Ontologic, Inc. ONTOS is a distributed database and has a client-server architecture. The server side manages the data storage. The client side provides the user interface and manages the mapping of data to the application process's virtual memory space. An ONTOS database may be contained on a single host or distributed over a number of hosts in a network. The database is controlled by a primary server which may be distributed over other hosts as well. The task of the server process is to manage the underlying storage of its portion of the database and respond to client requests over the network. The primary server of each database maps objects to their respective servers. It is

also responsible for all operations global to the database. These operations include database open and close requests and the control of multi-server commit. The client is implemented as a function and a class library that is linked into the application process. It manages the communication between applications and one or more servers. The interface between the application and the client itself is composed of a relatively small set of functions and classes.

ONTOS's object model uses objects as the basic modeling construct. The ONTOS class library provides a root Object class which is the parent of all persistent classes. Object defines a constructor and a destructor for creating and deleting objects in the object database. ONTOS objects can be referenced by name or by reference. Aggregate classes are supported for grouping objects and for modeling one to many relationships. The aggregate classes provided by ONTOS include Dictionaries, Sets, Arrays and Lists. The query processing in ONTOS adds navigational extensions to SQL predicate calculus semantics, and enhances the SQL's notion of table by including arbitrary collections of objects such as aggregates.

The ONTOS Object SQL extends the SELECT, FROM, and WHERE clauses of standard SQL to provide a query facility for the existing user base. The ONTOS Object SQL also provides an ad hoc query mechanism, aggregate manipulation, and convenient persistence for object language systems. In ONTOS, the FROM clause accepts class names and any argument that evaluates to a collection of objects. The SELECT clause accepts property names, member function invocations, and "navigational style" property path expressions. The WHERE clause is extended to allow arbitrary Boolean expressions.

A.4.7 O_2

The O_2 system is an OODBMS developed by the O_2 Technology[8, 9, 141]. The O_2 system consists of three tightly integrated components: O_2Engine, O_2Development Environment, and O_2Tools. O_2Query is an integral part of the O_2Development Environment. O_2Query provides a query language used to manipulate O_2 objects. It can be used interactively for ad hoc access to the object database, from a programming language, or from the O_2C 4GL. There were a number of design chocies made in the design of the O_2 query language. First of all, the query language violates encapsulation in its ad hoc mode, but not in its programming or embedded mode. Secondly, a query returns an object or a value. Returned objects are those already existing in the database while new values can be built by a query. Thirdly, the query language is functional in nature. Finally, the query language ignores types and the type hierarchy. Type checking is performed at run time in the query mode and at compile time in the programming mode.

O_2 handles values and objects. Data can be either objects with identify and encapsulation, or complex values manipulated through algebraic operations. Objects are instances of classes, and values are instances of types. Each object has an object identity which is unique, value independent, and allows reference to the object. An object belongs to a class which is characterized by the type of its instances and by a set of operators called methods. An object can only be manipulated via the methods given in its class interface. O_2 also supports late binding for methods. In O_2, values are characterized by their types. Values may be composed of objects as objects may be composed of values. However, objects are encapsulated through class interfaces while values are not. Values have no identity. Standard operators are available for

manipulating **tuple**, **set** and **list** values.

The O_2 programming interface provides a schema command language for defining database schemas. The query language provided by O_2 is an SQL-type functional language that offers associative access to the object database through the SELECT-FROM-WHERE construct. The semantics of an O_2 query $f(x_1, x_2, ..., x_n)$ is defined as follows: for every database, f defines a partial mapping $(O \cup V)^n \mapsto O \cup V$, where O is the set of persistent objects of the database, and V is the set of the persistent and possible values of the database. Possible values of the database refer to those values whose components are atomic values or objects belonging to O. The SELECT-FROM-WHERE construct is a set filter. The traditional semantics are preserved. The FROM clause indicates the filtered set. The WHERE clause specifies the conditions of the elements on which the operations expressed inside the SELECT clause will be executed. O_2 provides an **element** operator to extract the unique element of a singleton set. **List** elements are ordered and can be accessed directly without filtering the entire list. To access all levels of a structure, e.g. to navigate through embedded sets or lists, O_2 provides a **flatten** operator that returns a set that is the union of the sets returned from an embedded query. The **define** keyword introduces named queries. A named query may denote a value or an object. A named query can be parameterized and denote a function. The **set** operator builds a set value. The **tuple** operator constructs a tuple value. Nested values can be built by combining the various operators. The only restriction is that the set operator must be applied on sets. Co-existence of objects and values in the object model allows users to use a complex value without defining a new class. This avoids undesirable growth of the class hierarchy. The notion of values in O_2 is similar to the concept of a composite

object in ORION or the concept of own attribute in EXODUS.

A.4.8 BKS Software

POET (Persistent Objects and Extended database Technology) is an ODBMS developed by BKS Software. POET has a tight semantic integration with C++. Database functionality like queries, transaction, indexing, and so on, is supported through extensions to C++. The goal of POET is to integrate object-oriented programming with database facilities. POET follows the object model of C++. Classes and objects are used along with common object-oriented features like encapsulation, inheritance, polymorphism, user-defined data types, and relationships among objects. Object persistence is achieved through class libraries and extensions to C++.

In POET, a class is made persistent by using the **persistent** keyword in the declaration of the class. The class definitions must be processed by the PTXX precompiler to produce the database. Then, objects can be created to populate the database. When the PTXX precompiler encounters a type declaration it creates a set called **AllSet** which holds all objects of that type. The **AllSet** can be iterated sequentially to access each object of a given type. POET supports value-based queries. POET queries always operate on a set and the query result is also a set. The POET query facility is provided through class library functions. The POET pre-compiler generates a query class for each class whose instances may become persistent. Conditionals and operators appear as methods of the query class. POET supports and extends a single programming language, C++, to accommodate database functionality without losing the flexibility of the object-oriented programming paradigm. The drawback of POET's approach is that the database functionality is provided through class li-

braries. For this reason, POET users must be familiar with C++ in order to develop applications with POET.

A.4.9 Statice

Statice is an ODBMS that runs with the Genera operating systems on Symbolics workstations[109]. Statice is written in Symbolics Common Lisp which includes the Flavors object-oriented programming language extension. Statice provides client programs with persistent, shared storage for data. The Statice object model is similar in some respect to Daplex and the object model of Iris. Objects are modeled by entities in the database. The Statice type system maps directly to the Common Lisp type system. The basic modeling construct is an entity which is an instance of an entity type. An application object model is composed of a set of entity types. An entity type has attributes that model both the properties of entities, and the relationships among entities. The attributes can be single-valued or set-valued. The type of an attribute can be an entity type, a conventional value type, or a user-defined value type.

The programmatic interface of Statice is closely integrated with Symbolics Common Lisp and with its object-oriented programming system. A database schema is defined in Common Lisp. An accessor function is defined for each attribute of an entity type. An accessor function retrieves the value of an attribute of an entity. The for-each special form provides associative access to entities in a database. It iterates over entities of an entity type or in a set and selects based on attribute values. This serves as the query language of Statice.

A.4.10 Unisys

SIM is a commercially available ODBMS from Unisys[56]. It is based on a semantic data model. The goal of SIM is to allow the semantics of data to be defined in the schema and make the DBMS responsible for enforcing the data integrity. SIM provides a rich set of constructs for schema definition and a non-procedural data manipulation language as a user query interface. The language constructs also allow the specification of inter-object relationships, integrity constraints, and generalization hierarchies modeled by directed acyclic graph.

In SIM, the primary unit of data encapsulation is a class which represents a meaningful collection of entities. The notion of subclass is supported. Subclasses inherit all the attributes of all their parent classes in the generalization hierarchy. Every base class has a system-maintained attribute called its surrogate which is used in the implementation of generalization hierarchies and entity relationships. *SIM distinguishes between data-valued attributes and entity-valued attributes.* A data-valued attribute describes a property of each entity in a class by associating the entity with a value or a set of values from a domain of values. An entity-valued attribute represents a binary relationship between the class that owns it and the class it points to. SIM also supports inverse relationships which can be explicitly specified by the user.

The DML of SIM is its database query language. The DML of SIM consists of a RETRIEVE clause, some aggregate functions like AVG, COUNT, and update statements like INSERT, MODIFY, and DELETE. A query is formulated to access instances of a class which is referred to as a *perspective class* in SIM. Instances of other classes may also be queried based on their relationships with the perspective class. The

links between the perspective class and other classes are established by applying qualifications on the attributes. The basic SIM query has the following construct:

```
[FROM <perspective class list>]
RETRIEVE [ TABLE [DISTINCT] | STRUCTURE ] <target-list>
[ ORDERED BY <order list> ]  [ WHERE <selection expression> ].
```

Perspective class list is the list of perspective classes for a query with optional associated reference variables. Target list and order list are lists of expressions made up of constants, and attributes of the perspective classes.

A.4.11 UniSQL

UniSQL Inc. has developed a suite of integrated ODBMS and application development products for object-oriented development, integration of multimedia data, and multidatabase access to RDBMSs and ODBMSs[143]. An object SQL is provided as part of the user interface. UniSQL offers a generalized and extensible relational/object-oriented model. In UniSQL, the traditional relational model and SQL can be used to build relational databases. In addition, object-oriented features are provided through extended support for nested tables, registered procedures (methods), and class (table) inheritance.

Nested tables are a technique for defining a field as being a row instance in another table. The notion of defining a field's data type as a row instance or a set of rows in another table is referred to as an arbitrary data type in UniSQL. To navigate the nested tables, the SQL syntax is extended to incorporate path expressions to identify the target nested column. In the case of **set-of** relationships, a special nested

cursor mechanism is used to step through the one-to-many relationship. Procedures (methods) are programs written in programming languages and are associated (registered) with a table. Procedures can manipulate data in the associated table, access other tables, request operating system services, or call other programs and procedures. Procedures are designed to shield users from the complexity of manipulating complex data. Inheritance is used to implement subtypes. A table can be declared as a subtype of its parent table. The child table inherits all of the attributes (columns) and registered procedures of the parent's table. UniSQL has extended the SQL syntax to include an ALL qualification in the SQL FROM clause. The ALL qualification instructs UniSQL to search subclasses as well as the parent table.

A.4.12 VERSANT

VERSANT is an ODBMS developed by the VERSANT Object Technology Corporation[144]. VERSANT is implemented within the VERSANT Scalable Distributed Architecture (VSDA). VSDA supports transparent data distribution, workgroup functions, and dynamic schema management and evolution. VERSANT provides a C++ application toolset and an object SQL. VERSANT Object SQL can be used interactively, or be embedded in C++.

VERSANT follows the C++ object model and uses objects and classes as the basic modeling constructs. Multiple inheritance is supported. In VERSANT, the object link is declared by <type>_link. <type>_link is a generated class used to hold the linked object. The link class results in a level of indirection. However, the traversal along the object links is made transparent. An attribute can also be defined as an aggregate link, <type>_agg. The deletion of the parent instance will result in the

deletion of the instance linked by the aggregate link. An attribute can be defined to be an array of links or embedded objects. The DDL and DML of VERSANT can be either C or C++.

VERSANT Object SQL is an SQL-based DML. Syntactically, the general structure of SQL is used including four basic data manipulation statements: SELECT, INSERT, UPDATE and DELETE. Like other OQLs, VERSANT Object SQL is semantically richer than relational SQL. User-defined methods can be used within the Object SQL statements. The OSQL SELECT statement follows the same basic structure of a relational SELECT statement:

```
SELECT <method or attribute> FROM class WHERE search_condition
GROUP BY method  ORDER BY method
```

VERSANT OSQL supports built-in aggregate functions like MIN, MAX, AVG, SUM, etc. VERSANT OSQL also provides an "intelligent pointer" mechanism that initiates automatic search through the subclasses of a given class. Data from multiple classes can be joined in three ways: flat join, subselects, or subqueries. Flat joins are structured like the traditional relational join. Subselects and subqueries allow join of data from two separate queries. Query result can be bound to program variables through a dynamic casting mechanism, called the OSQL_AS() function.

A.5 Survey of Proposals and Research Prototypes

A.5.1 AT&T

CQL++ is a declarative front-end to the Ode OODBMS[39]. Ode (Object database and environment) is a research prototype developed at the AT&T Bell Laboratories[2]. A database programming language, O++, is provided to define, query, and manipulate the object database. O++ borrows and extends the object model of C++. CQL++ hides users from knowing O++ details such as object identifiers, public and private members of objects, and the implementations of member functions. CQL++ queries operate upon sets of objects and return sets of objects.

In Ode, a database is a collection of persistent objects. The basic modeling construct is objects. At the programming level interface, such as that provided by O++, an object consists of an object identifier and a state. On the other hand, CQL++ provides a higher level interface to Ode in which users can manipulate objects directly without explicit use of object identifiers. Although Ode follows the O++ object model, CQL++ does not provide the full power of the O++ object model. CQL++ distinguishes between persistent and transient objects. Persistent objects reside in the database and are grouped into clusters. Transient objects can be placed in temporary clusters.

The persistence in O++ follows a number of principles. First of all, persistence should be a property of object instances instead of types. Secondly, it should be possible to allocate objects of any type in either volatile or persistent store. Thirdly, there should be no difference between accessing and manipulating persistent and volatile objects. Finally, it should be possible to move objects from persistent store to volatile store in much the same way as it is possible to move objects from the stack

to the heap and vice versa. CQL++ is based on an type-independent object algebra that preserves the closure property of SQL. The classes and objects created using CQL++ can be intermixed with those created with O++.

CQL++ combines an SQL-like syntax with the C++ object model. CQL++ is designed for SQL users as far as the language syntax is concerned. CQL++ is based on an object algebra that has the closure over sets of objects. Each CQL++ statement or operator takes sets of objects as input and returns sets of objects. CQL++ queries are allowed to be nested due to the closure property. A CQL++ query has the following basic form:

```
SELECT <projection-list> FROM <collections:clusters or sets>
                         WHERE <search-expression>
```

Each operand in the FROM clause is associated with a range variable. Set-valued attributes may be used as operands in the FROM clause. The SELECT statement returns a set of objects constructed from the cross product of collections that satisfy the search expression. The returned portions of selected objects correspond to the projection list in the SELECT clause. CQL++ provides clusters and sub-clusters as a way of grouping objects of the same type. In other words, CQL++ supports explicit type extents. CQL++ also supports set variables and object variables to facilitate the manipulation of objects or sets of objects. Set variables and object variables can be used to reference sets or objects.

A.5.2 GTE

GTE Laboratories' Distributed Object Management (DOM) project has conducted research on distributed object management technology[75]. The object model of the DOM project requires language facilities to support multimedia data types[75]. A key issue in database language support for multimedia data types is the integration of database access facilities with the general-purpose programming facilities required for specifying operations of multimedia data types. In [75], the language requirements of the DOM project were described and illustrated in an extended SQL syntax.

A.5.3 University of Wisconsin (EXODUS)

The EXTRA data model and the EXCESS query language are part of the EXODUS extensible database system[26, 27, 28]. EXODUS has been designed as a toolkit for use as a basis in constructing a spectrum of target database facilities. The functionality of EXCESS aims at supporting associative access to object databases. EXCESS is designed to be a query language that can be used for both business and engineering data. EXCESS is an associative query language that is amenable to query optimization techniques. In the EXTRA data model, a database is a collection of named persistent objects. EXTRA separates the type definitions from the declarations of their instances. Tuple, set, and array are provided as type constructors that can be composed arbitrarily to form new types. User-defined ADTs and multiple inheritance are also supported.

EXCESS allows users to collect related objects in semantically meaningful aggregates which can then be queried. EXCESS is based on QUEL[101]. EXCESS provides a uniform syntax for formulating queries over sets of objects, sets of refer-

ences, and sets of values. An EXCESS query has the basic QUEL range-retrieve-where construct. Each range variable has an associated type, and the query may only refer to attributes associated with this type. The simplest form of the range statement has the traditional QUEL syntax:

range of <Variable> is <Range_Specification>

The range specification must identify either a named persistent set, or a subrange of an array. Arrays are treated as sets. An implicit range variable is provided for each set or array specified in the target list or in the qualification of a query. EXCESS also provides a path syntax to simplify the task of formulating queries over nested sets of objects. If one of the elements of a path is a single object, it is treated as a singleton set. If a query is embedded in a programming language, the object identifier can be bound to a host variable for subsequent manipulation. EXCESS supports three types of joins: functional joins, explicit identity joins, and value-based joins. A number of aggregate functions like AVG, MIN, etc. are also supported in EXCESS.

A.5.4 University of Wisconsin (EMS)

FOX (Finding Objects of eXperiments) is a declarative query language for a scientific experiment management system (EMS) being developed at the University of Wisconsin[114]. The goal of the EMS is to support scientists in managing their experimental studies and the generated data. An OODBMS is one of the components of the EMS. A data model, MOOSE (Modeling Objects Of Scientific Experiments), and a declarative query language, FOX, have been developed for the special needs of experimental sciences.

MOOSE is an object-oriented data model that supports complex objects, object identity, classes, and multiple inheritance. The basic modeling construct in MOOSE is an object. Three kinds of classes are supported in MOOSE: primitive, tuple, and collection. The primitive class is similar to the type literal in our reference model. Tuple and collection classes are created by users. There are two major relationships between MOOSE classes: connection relationships and inheritance relationships. A connection relationship between two classes implies a logical or physical relationship between their object instances. An inheritance relationship between two classes implies a semantic correspondence between their object instances.

The DDL of MOOSE provides constructs for creating, updating, and destroying classes and relationships. FOX is the DML for MOOSE. The basic structure of a FOX query is derived from SQL:

```
for <range-binding-list> select <projection-list>
where <qualification> as <name>;
```

Each query or sub-query has an optional naming clause, "as <name>", which attaches a name to the query result. Therefore, a query result becomes a named object. Named objects allow users to access and reuse the query result. In the for clause, object variables may be bound to the members of class extents or collection objects described by path expressions. The select clause allows users to specify the structure of the query result as well as what to retrieve. Named objects of FOX can be persistently saved in the database. On the contrary, instance variables and object variables in Iris and CQL++ cannot be made persistent.

A.5.5 HP

OSQL is the object-oriented database language developed for the Iris object-oriented DBMS at Hewlett-Packard Laboratories[44, 62]. Iris has a layered architecture which consists of a storage manager, an object manager, and a set of interfaces. The Iris Storage Manager is a conventional relational storage subsystem. The Iris Object Manager supports the Iris data model. Interactive and programming interfaces are provided for access through the Iris Object Manager which in turn interacts with Iris Storage Manager to fetch the data.

The Iris DBMS is based on a semantic data model that has three major constructs: objects, types, and functions. Objects can be literal objects or surrogate objects. A surrogate object has a system-generated unique identifier. Examples of surrogate objects are types, functions, and user objects. *An object may gain and lose types dynamically.* Attributes of objects, relationships among objects, and computations on objects are expressed as functions. In other words, Iris object semantics are fully determined by the behavior of functions. Iris supports three methods of function implementation: stored, derived, and foreign. The extension of a stored function is maintained as stored data. Derived functions are computed by evaluating an Iris expression. A foreign function is a subroutine written in some general-purpose programming language. Iris queries are expressed in terms of functions and objects. Queries are compiled from their object representation to a relational algebra representation which is then used to access the database or to invoke foreign functions to access other data sources.

Iris OSQL has been implemented as an interactive interface as well as a language extension embedded in programming languages. The application object model can

be exposed to the programming language through the interfaces provided by the Iris Object Manager. Extensions have been made to programming languages to add the notion of object persistence. Persistent-CLOS (PCLOS) is an example.

Iris OSQL has three major extensions to SQL. First of all, users manipulate types and functions rather than tables. This gives Iris OSQL a functional flavor. Second, objects may be referenced directly through their keys. Third, user-defined functions and Iris system functions may appear in WHERE and SELECT clauses. The major functionality of the Iris OSQL includes associative retrieval, bulk update, the support of cursor, and the integration with programming languages. Instances of Iris types can be bound to host variables. The property of Iris objects are defined in terms of functions. The SELECT and CURSOR statements return Iris objects. SELECT returns all objects that satisfy the search criteria. CURSOR provides control over how the result of a query should be returned. Attributes of Iris objects are accessed through functions. The use of in-line definitions of property functions; i.e., functions of one argument, simplifies the definition of types and the initialization of objects.

A.5.6 Brown University

Sarkar and Reiss[93] proposed a rule-based object query language called OQL. The motivation of their research is to use the OQL and an ODBMS to construct abstract information about programs, and then allow the program database to be queried. The result of the queries are program abstractions that can be visualized graphically in a program visualization system. OQL is designed to be an ad hoc query language.

The data model supports four basic types: Integer, Float, Boolean, and String.

The domains of basic types are called basic domains. Each basic domain consists of a countably infinite set of atomic values. Structured values are built from atomic values. Only structured values can be values of objects. Each object belongs to a class. A class is defined by specifying a name, a superclass, a structure, and methods.

OQL is rule-based, statically typed, and allows stratified negation. A program consists of a sequence of statements. Each statement is either an assignment or a rule. An assignment assigns an r-value to an l-value if the qualifiers are satisfied. The object expression in the assignment statement provides an r-value, and the path expression provides the l-value. The symbol ":=" denotes the assignment operator. A rule has a head and a body. A head is a special type of literal with a path expression, an optional equality operator, and an object expression. There are three system-defined equality operators for id-equality, shallow-equality, and deep-equality. Each body consists of a set of literals. Each literal is either a generator or a qualifier. There are set-valued and list-valued objects. A generator has a range variable. A qualifier specifies a condition. A class expression represents a set of objects of the class. OQL programs are translated into algebraic operations, assignment operations, and REPEAT_UNTIL loops.

A.5.7 ORION

ORION is a research prototype developed as an ODBMS at the Microelectronics and Computer Technology Corporation[10, 65, 66, 67]. Advanced features supported in ORION include versions and change notification, composite objects, dynamic schema evolution, transaction management, queries, and multimedia data management. An object subsystem is provided to support most of the above-mentioned

features. The storage subsystem provides access to object on disk. The transaction subsystem manages concurrency control and recovery. A message handler receives and processes all messages sent to the ORION system.

In ORION, objects are the basic modeling construct. Each object has a unique object identifier, and encapsulates a state and a behavior. The state of an object is the value of the object's attributes. The behavior of an object is the set of methods that operate on the state of the object. The relationship between an object and its class is the instance-of relationship.

An ORION query can be a single-operand query or an n-operand query. A single-operand query can be an acyclic query or a cyclic query. Complex queries can be formulated by combining more than one simple queries with set operations. The general form of a simple query is as follows:

```
SimpleQuery ::= select TargetClause |
      select TargetClause from RangeClause |
      select TargetClause where QualificationClause |
      select TargetClause from RangeClause where QualificationClause
```

The syntax of a query consists of three clauses: target, range, and qualification clauses. The target clause specifies the attributes to be retrieved. The range clause specifies the binding of variables, called object variables, to the corresponding sets of instances of classes. The qualification clause specifies the qualification conditions as a Boolean combination of predicates. The query language of ORION supports both object equality and value equality. The comparator for object equality is denoted by "=", while for value equality it is denoted by "==". *A single-operand query retrieves*

objects from only one target class. Recursive queries are supported in ORION. A path segment can be defined recursively:

(<path-1> (recurse) <path-2>) <comparator> <value-or-variable>

The number of repetitions of **segment** is limited by the maximum length of the path expression. Two types of methods may appear in an ORION query: a derived-attribute method or a predicate method. A derived-attribute method computes a value from the attribute values of the object or some other objects in the database. A predicate method is used as a predicate and returns the logical constants **True** or **False**. The value returned by the predicate method can then participate in the evaluation of the Boolean expression in a query. Multiple-operand queries may result from the inclusion of user-specified join attributes or the set operations. To allow joins of classes on user-specified join attributes, the class hierarchy must be accounted for in the scope of query evaluation. The general form of ORION queries that allow set operations is shown below:

```
Query ::= (Query) |
          Query Union Query | Query Intersection Query |
          Query Difference Query | SimpleQuery
```

The operand of a set operation is a set of instances which may be the set of instances of a class defined in the database, or a set of instances obtained as the result of a query.

A.5.8 University of Zurich

Quod is a declarative DML developed for NO^2 (New Object-Oriented data model)[49]. NO^2 is the data model of CoOMS (Combined Object Management System). CoOMS is a structurally object-oriented database system being developed at SNI (Siemens-Nixdorf Informationssysteme). It is intended to serve as a DBMS and as the database component of the ITHACA kernel. ITHACA (Integrated Toolkit for Highly Advanced Computer Applications) is an ESPRIT project.

In NO^2, the basic modeling construct is objects. Objects are distinct from values which are used to describe properties of objects. Objects are instances of NO^2 types which are actually object types. NO^2 contains a collection of basic value sets: INTEGER, REAL, FLOAT, STRING, CHARACTER, BOOLEAN, and LONG FIELD. NO^2 distinguishes between objects and values. Complex objects and is-part-of relationships are supported. Complex values can be created using a set of orthogonal constructors: LIST, SET, TUPLE, and ARRAY. Complex object structures can be defined either by including objects as components into other objects, or by referencing them. Object types are organized in a specialization/generalization hierarchy which allows multiple inheritance of structural properties.

NO^2 supports a data definition language for defining database schemas based on its data model. An SQL-style query language, Quod, is supported in NO^2 for querying and manipulating objects in the object database. Basic NO^2 queries follow the SELECT-FROM-WHERE construct. The SELECT clause specifies which parts of the objects or values identified by the FROM and WHERE clauses are to be retrieved. In NO^2, the result of a query is existing objects, or values of objects. To create new objects from existing objects, a SELECT statement must be explicitly nested in an

INSERT statement. The SELECT clause can also be used to define the structure of query results. Queries can be embedded in the SELECT clause such that the query result can be used in another query. The FROM clause specifies extensions of object types. The WHERE clause specifies the restrictions that have to be satisfied by the objects or values specified by the SELECT and FROM clauses. Path expressions can be used to access objects. Operators similar to those found in O_2Query are provided to allow access to values or complex values. Built-in operators are provided for sorting or grouping lists. A special feature of Quod is the support of a mechanism for defining parameterized, named queries which can then be incorporated in other queries. The syntax of such queries is as follows:

define query <name> [(<parameters>)] as <query>.

The query identified by <name> may occur anywhere a query is allowed. Recursive queries are supported in Quod in the form of the construction of transitive closures. Quod also provides constructs for the insertion, update, deletion, and migration of objects. The migration of an object means a change of the type of an object without explicitly creating a new object. The identity of the object remains the same while its value changes, and the object is added to a new type extension after being removed from its old type extension.

A.5.9 University of Alberta

TIGUKAT is a research prototype for object data management developed at the University of Alberta[86]. Existing components of TIGUKAT include an object model, a query model, an object calculus, an object algebra, a data definition language (TDL), a query language (TQL), and a control language (TCL). The

TIGUKAT object model is built on top of the EXODUS storage manager.

The TIGUKAT object model is defined behaviorly with a uniform object semantics. All access to and manipulation of objects are based on the application of operations on objects. All information is modeled by objects and has the status of a first-class object. The object model provides a primitive type system. The notions of type and class in TIGUKAT follow our reference object model. The TIGUKAT query model is a uniform extension of its object model. The query model defines a logical object calculus, an equivalent behavioral/functional algebra, and an SQL-like query language. The object calculus defines predicates on collections of objects and returns a collection of objects.

The TIGUKAT user language has three parts: TDL, TQL, and TCL. The main function of the TIGUKAT user language is to support the definition, the manipulation, and the retrieval of objects in a TIGUKAT object database on an ad hoc basis. TDL supports the definition of metaobjects. All type, collection, class, behavior, and function objects are considered metaobjects. TDL is logically divided into six groups of statements: type declaration, class declaration, collection declaration, behavior manipulation, function declaration, and association. For example, the general syntax of the type declaration statement is given below:

<type declaration>: **create type** <new reference> **under** <type list> <behavior list>

TQL allows the retrieval of objects in an object database. TQL has four basic operations: **select**, **insert**, **delete** and **update**. Each of these operations operates on a set of input collections, and returns a collection. The basic query statement in TQL is the **select** statement which has the following general syntax:

```
<select statement> : select <object variable list>
                     [into [persistent [all] ] <collection name>
                     from <range variable list> [where <boolean formula>]
```

The select clause identifies the objects to be returned in a new collection. There can be one or more object variables in this clause. The object variables can be in the form of simple variables, path expressions, index variables, or constants. The into declares a reference to a new collection returned as the result of a query. The from clause declares ranges of object variables in the select and where clauses. The where clause defined a boolean formula which must be satisfied by the objects returned by a query. TCL supports session specific operations.

A.5.10 UniSQL

XSQL was proposed in [64] for querying object-oriented databases. The language extends path expressions and adapts the first-order formalization of object-oriented languages. The goal is to make XSQL easier to use and have greater expressive power. The approach of XSQL is based on F-logic[63] in order to give precise semantics to XSQL without violating encapsulation.

The basic modeling construct in XSQL is objects. An object has a logical identifier which does not have to be unique. A physical object identity is unique and represents a surrogate or a pointer to an object. Objects are described via attributes. All XSQL objects are tuple-objects. Each entry in a tuple-object is the value of one attribute. Set-objects are described as tuple-objects with a single, set-valued attribute. *An operand appearing in a query may be an attribute name or an object name.* This feature is used to help users formulate a query without knowing the

structure of the database. Classes are used to group related objects. An object is related to a class through the instance-of relationships. The subclass or IS-A relationship is defined between classes. The type of a class is determined by the types of its methods. The type of a method in a class is described as a signature. Methods defined in the scope of a class are inherited by each of the subclasses of the class and by all of its instances. The same holds for attributes.

Path expressions describe paths along the composition hierarchy. A path expression can be viewed as a composition of methods. The general form of a path expression can be described as:

$$\mathsf{selector}_0.\mathsf{AttEx}_1\{[\mathsf{selector}_1]\}.\ \ldots\ .\mathsf{AttEx}_m\{[\mathsf{selector}_m]\},$$

where $m \geq 0$, and braces denote optional terms. A selector is either an object identifier or a variable. The attribute expression, AttEx_i, is either an attribute name or an attribute variable that ranges over attribute names. *A database path is any finite sequence of database objects. A path expression describes a subset of the set of all database paths. This subset is determined by the semantics of the path expression.* Path expressions can be used in a SELECT-FROM-WHERE query construct. Path expressions may appear in the WHERE clause to identify the set of objects to be retrieved. XSQL also supports class variables. The names of class variables are prefixed with the "#" sign. Path expressions can be compared via the comparators like =, !=, >, etc. These comparators are modified with existential and universal quantifiers since path expressions represent sets.

A.5.11 Texas Instrument

ZQL[C++] is an extension of C++ with object query capabilities[18]. An initial prototype of ZQL[C++] was built as part of the Zeitgeist OODB project at Texas Instruments and further development has continued on the Open OODB system[111]. A commercial implementation by VERSANT Object Technology based on ZQL[C++]'s specification is available.

The object model of ZQL[C++] is the type system of the C++ programming language. ZQL[C++] supports explicit sets; i.e., collections of objects can be defined and maintained explicitly by an application. Collection types are defined using C++'s parameterized classes. Query processing of Open OODB is based on a logical algebra and a set of execution algorithms that defines the query evaluation environment[17]. The foundation of the logical algebra includes the traditional set and relation operators.

The design of ZQL[C++] achieves integration between C++ and query capability. A uniform type checking can be done for the entire application including queries and programming language expressions. Multiple sets of a type may co-exist in an application. A query optimizer has been built for the Open OODB system[17]. The goal of ZQL[C++] includes allowing queries on transient or persistent data, permitting user-defined functions in the formulation of queries, supporting data abstraction and inheritance, and providing support for queries on semantically different collection types and complex objects. A ZQL[C++] query is an extension of the SQL query block:

<result> = SELECT <objects> FROM <range variable> IN <collection> WHERE <predicate>;

The SELECT clause identifies the type of the objects in the collection to be returned by the query. The type may be a new type or the same type as one of the range variables. The FROM clause declares the range variables and the target collection to be queried. Several variables ranging over several collections can be declared. Pointer range variables are allowed. The WHERE clause specifies the predicate that defines the properties to be satisfied by the objects to be retrieved. The predicate can be any conditional expression that is legal in a C++ if statement. Path expressions may be single-valued or set-valued.

A.5.12 University of Florida

An object-oriented query language, OQL, was developed at the University of Florida. A graphical user interface implemented for OQL was reported in [106]. A query can be specified through browsing and traversing in the graphical user interface. OQL is based on an object-oriented semantic association model (OSAM*). A database schema is represented as a network of associated object classes which form a semantic diagram. Domain object classes (D-class) model the domain of primitive data types; e.g., integers, strings, and so on. Entity object classes (E-class) model application objects. The query model is based on OSAM* and a set of query semantic rules.

OQL has its own runtime environment. There were no programming languages integrated with OQL. The schema diagram can be created and browsed via a graphical user interface. It was not reported how the database is populated. No separate type definition language is needed. An OQL query returns a subdatabase. A subdatabase consists of an intensional association pattern and a set of extensional

association pattern. An intensional association pattern is represented as a network of E-classes, their associations and descriptive attributes. An extensional association pattern is a network of instances and their associations that belong to the classes and association types of the intensional association pattern. OQL allows users to specify the desired subdatabase by specifying its intensional association pattern, its extensional association patterns and the operations to be performed on the classes of the subdatabase. OQL preserves the closure property since the result of an OQL query is structured and modeled by the same data model, OSAM*. A query block in OQL is very similar to the SELECT-FROM-WHERE construct. The context clause specifies the intensional and extensional association patterns via the association pattern expression. The association operator and set operators are provided to construct subdatabases. A subdatabase can be assigned to a variable and be saved permanently.

A.6 Standardization Issues

Standardization facilitates the development of interoperable systems. In the context of ODBMSs, standardization activity is on-going in the areas of object model[45, 123], object data management[45, 122], and object query language[29, 46, 146]. In this section, we describe the two major efforts in the standardization of OQLs: SQL3 and ODMG.

A.6.1 SQL3

National and international SQL standardization committees have been working on the extensions of SQL to meet the requirements of managing complex objects

in engineering and multimedia environments[146]. These extensions include object identifiers, abstract data types, inheritance hierarchies, and other features normally associated with object data management. This extended version of SQL is commonly referred to as SQL3. The historical definition of an object is a row in a relational table. SQL3 carries this concept forward and adds two new types of rows. Therefore, we have three types of objects in SQL3: a row in a table with no object identifier, an SQL3 row with an object identifier not visible to users, and an SQL3 row with object identifier visible to users as the first column of the table containing the row. An object type can be defined as an abstract data type (ADT). An ADT definition includes the definitions of attributes, operations, subtypes, and object identifier. Common object-oriented features like encapsulation, subtypes, inheritance, and polymorphism are supported in SQL3.

Currently, all object manipulations in SQL3 are achieved through table operations. SQL3 allows specification of a "tabular" shell over an ADT class. Constructor and destructor functions are automatically invoked when rows are inserted or deleted from the table. The table itself is the collection of all objects. Therefore, SQL query and update statements may then be applied to the table without any language enhancements. SQL3 does not describe how to integrate with programming languages.

SQL3 is a computationally complete language for the definition and management of persistent objects. SQL functions are completely defined in SQL. External functions have their interface definitions specified in SQL and allow their implementations to be written in programming languages. The control structures of SQL3 include compound statement, exception handling, and flow control statements.

A.6.2 ODMG

The Object Database Management Group (ODMG) is a consortium of ODBMS companies whose goal is to establish an industry-wide agreement for object-oriented database technology. ODMG-93, the Object Database Standard, is the result of the work by ODMG[29]. The object model of ODMG-93 has been described in Section A.3.1. Since ODMG-93 is only a standard specification, how the object model is supported in a specific ODBMS will depend on the actual implementation. In ODMG-93, a query consists of a set of query definition expressions followed by an expression. The semantics of the query model has been described in ODMG-93 although the query processing remains an implementation-dependent issue.

ODMG-93 has an ODL and an OQL. ODMG-93 ODL is a specification language used to define the interface to object types that conform to the ODMG object model. ODL is independent of any programming language. It is also compatible with the OMG's Interface Definition Language (IDL). Therefore, the application object model defined by ODL can be shared by multiple programming languages. How a specific programming language is able to use the object types defined by ODL depends on the binding of that programming language to the ODL.

ODMG-93 OQL is not computationally complete. It has an abstract syntax and provides declarative access to an object database. ODMG-93 OQL has one concrete syntax which is SQL-like. Another concrete syntax may be defined for merging the query language into programming languages. The OQL does not provide update operators. It relies on operations defined on objects for updates. The semantics of ODMG-93 OQL allow a collection to be constructed from a set of query expressions. This implies that a heterogeneous aggregate can be constructed. Since the

constructed aggregate can then be queried, it is possible to issue a query against several type extents.

A.6.3 Analysis

The standardization of OQLs sparks a number of interesting issues especially when the standardization activities appeared to be an aftermath since many OQLs have already been implemented. Several ODBMS vendors have committed to either implementing or adjusting exist implementations to accommodate the OQL proposed by ODMG[29]. There are also companies who have started to adopt SQL3 in the development of multimedia data management[89]. Although we have no clue as to which standard will prevail or whether both standards will co-exist, it is worthwhile to explore a number of standard issues related to OQL standards and existing OQLs. For example, some existing OQLs take the approach of extending the SQL. It is interesting to know what SQL features are supported and what extensions have been made. Then we can evaluate whether the OQL will be able to accommodate or at least interoperate with existing OQL standards. Furthermore, the standardization of OQLs will have an impact on the integration and interoperation of distributed heterogeneous DBMSs. How the existing OQLs will provide room for multidatabase integration and interoperation remains an open problem.

APPENDIX B. THE SPECIFICATIONS OF THE $\mathcal{ZMS}$ IN IDL

B.1 Introduction

In this appendix, we provide the high-level specifications of the $\mathcal{ZMS}$ written in CORBA IDL[122]. We intend to use these specifications to provide a complete description of the high-level $\mathcal{ZMS}$ design. There is a direct correspondence between the $\mathcal{ZMS}$ class hierarchy and the IDL specifications. Each IDL interface definition corresponds to a class definition in the $\mathcal{ZMS}$ class hierarchy. IDL is declarative, object-oriented, and independent of programming languages. The declarativeness, object-orientation, and simplicity of IDL make it appropriate for specifying the $\mathcal{ZMS}$. Figure B.1 shows the major components of a CORBA IDL implementation. The IDL front-end generates abstract syntax graphs on behalf of IDL source files. Abstract syntax graphs are independent of programming languages. The emitters of the IDL back-end take the abstract syntax graphs and generate output stubs bound to the programming language associated each emitter; e.g., the emitter for the C programming language will generate output stubs as C header files and C source files. How the IDL specifications lead to the actual system implementation can be summarized as a sequence of mappings:

$$\overbrace{class \xrightarrow{1:1} IDL}^{PL-independent} \underbrace{\xrightarrow{1:M} \langle stub, skeleton, template\rangle}_{PL-specific} \overbrace{\xrightarrow{1:N} \langle stub, skeleton, expanded-template\rangle}^{Implementation-dependent}$$

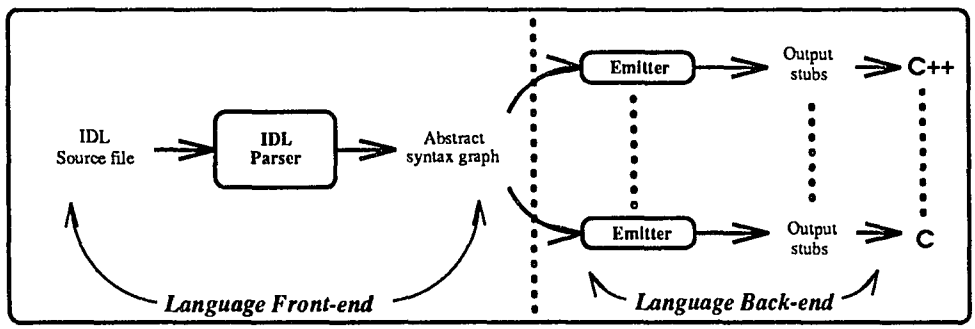

Figure B.1: CORBA IDL.

The **stub**, **skeleton** and **template** files are commonly referred to as the IDL output stubs. The skelton files are used by the template files. The stub files are used by the client programs that need to use the class specifications defined by the stub files. The template files are expanded to provide implementations for the operations defined in the IDL source files. The same template file may be expanded in different ways. The resulting expanded template can be compiled and linked with the client programs that have included the corresponding stub files. For the remainder of this appendix, we will provide IDL specifications for major $\mathcal{ZMS}$ components and describe our design in the context of these IDL specifications.

B.2 ZVS Object, ZMS & ZMS Manager

All the runtime components of the $\mathcal{ZMS}$ are instances of subclasses of ZVSObject. The common attribute, LCE, provides links to the local computing environment hosting the runtime component. The two common operations, **startup()** and **shutdown()**, are defined to determine how to start and stop the runtime component in a controlled manner.

```
// IDL source file: ZVSObject.idl
#include <ZVSLCE.idl>
#include <ZMSGlobalDecl.idl>

interface ZVSObject
{
   readonly attribute long   ID;
   readonly attribute ZVSLCE LCE;          \\ point to a ZVSLCE object
   readonly attribute ZVSServiceType ST;

   ReturnCode startup();
   ReturnCode shutdown();
};
```

A ZMSystem object coordinates the initialization and termination of a multidatabase system. During the $\mathcal{ZMS}$ initialization, the system configuration is loaded and major $\mathcal{ZMS}$ runtime components are started in a pre-specified sequence. The system configuration is defined in a SysConfig structure. The operation, Load(), is called to load the system configuration to the main memory. Once the system is started, a simple command interface can be called up to manage the system configuration. The command table is saved in the structure pointed to by the attribute, CmdTbl. The command interface is invoked by the operation, ZMSCmdInterface(). When the $\mathcal{ZMS}$ is terminated or needs to be restarted, the operation, Save(), is called to update the system configuration files.

```
// IDL source file: ZMS.idl
#include <ZVSObject.idl>
```

```
#include <ZVSServer.idl>
#include <HostManager.idl>
#include <CooperatingAgent.idl>
#include <SysConfig.idl>
#include <ZVSManager.idl>

interface ZMSystem : ZVSObject, ZVSManager
{
   // attribute ZVSLCE LCE;          // inherited from ZVSObject
   // readonly attribute CommandTableEntry CmdTbl; // inherited
   // readonly attribute short          NumOfCmds; // inherited

   attribute ZVSServer        Server;
   attribute HostManager      HM;
   attribute CooperatingAgent CA;

   readonly  attribute SysConfig SC; // point to the SysConfig structure

   // startup() and shutdown() are inherited from ZVSObject
   ReturnCode Load();
   void       Save();
   void       ZMSCmdInterface();
 };
```

The system configuration information includes the lists of running $\mathcal{ZMS}$ components, the list of the $\mathcal{ZMS}$ view repositories, and the locations of system configuration files. The operation, LoadSysConfig, loads the system configuration to the main memory. The operation, SaveSysConfig follows the path defined in SysFilePath and saves the system configuration to the secondary storage. The operation, Management, starts a user interface that provides management facilities for managing the system configuration. ServerList is a subclass of List and ZVSServer. HostManagerList is a subclass of List and HostManager. CoopAgentList is a subclass of List and CooperatingAgent. Operations defined for List include common utilities like insert, remove, update, search, etc. The semantics of these operations are specialized in ServerList.

The same is true for both HostManagerList and CoopAgentList. It is possible to define a gereric list structure for all kinds of lists. However, we chose to define a generic list and specialize it for different kinds of lists to keep the specifications more readable.

```
// IDL source file: ZVSSysConfig.idl>
#include <ZVSLCE.idl>
// #include <List.idl>
#include <ServerList.idl>        // not included in the specialization
#include <HostManagerList.idl>   // not included in the specialization
#include <CoopAgentList.idl>     // not included in the specialization
#include <ZMSViewRepositoryList.idl>

interface SysConfig
{
   readonly attribute ZVSLCE                env;
   readonly attribute ServerList            Server;
   readonly attribute HostManagerList       HostManager;
   readonly attribute CoopAgentList         CoopAgent;
   readonly attribute List                  SysFilePath;
   readonly attribute ZMSViewRepositoryList ZVSRL;

   void LoadSysConfig(in env);
   void Management(in env);           // A management interface
   void SaveSysConfig(in env);
};
```

The $\mathcal{ZMS}$ has a client-server architecture. The interactions between clients and servers follow pre-established contract/protocol. Most of the servers in the $\mathcal{ZMS}$ are modeled as instances of the subclasses of ZVSManager. ZVSManager is an abstract class that provides a template for describing servers and the contract between clients and servers.

```
// IDL source file: ZVSManager.idl
#include <ZVSRequest.idl>
#include <ZVSLCE.idl>
#include <ZMSGlobalDecl.idl>

interface ZVSManager
{
   readonly attribute CommandTableEntry CmdTbl;
                                        // point to the command table
   readonly attribute short             NumOfCmds;
   readonly attribute ZVSServiceType    ST;

   ReturnCode Dispatch(in ZVSLCE     env,
                       in short      Destiny,
                       in ZVSRequest request);

   ReturnCode ReceiveRequest(in ZVSLCE env, in  ZVSRequest request);
   ReturnCode RespondRequest(in ZVSLCE env, out ZVSRequest response);
};
```

Some managers are implemented as multi-service network servers which may invoke or request services from other network servers on behalf of a request. The invocation is done through the operation, Dispatch. Each manager has a command table that defines the command syntax for client requests and server responses. Notice that in the context of the $\mathcal{ZMS}$, the client requests are referred to as the messages/requests from any system component that is able to send messages/requests to network servers.

B.3 ZMS Server

A $\mathcal{ZMS}$ server is an instance of ZVSServer. ZVSServer, has five subclasses which define the five subordinate managers of a $\mathcal{ZMS}$ server: ClientMgr, CoopAgentMgr, ThreadMgr, ViewMgr, and TransactionMgr. A ZVSServer object coordinates

the start-up and shutdown of its subordinate managers. As soon as the targets of incoming requests are identified, the requests are dispatched to appropriate subordinate managers for further processing. The attribute, Service, defines the specific service provided by the $\mathcal{ZMS}$ server or its subordinate managers. The attribute, Cell, identifies the administrative domain that hosts the the $\mathcal{ZMS}$ server.

```
// IDL source file: ZVSServer.idl
#include <ZVSObject.idl>
#include <ZVSManager.idl>
#include <ClientMgr.idl>
#include <CoopAgentMgr.idl>
#include <ThreadMgr.idl>
#include <ViewMgr.idl>
#include <TransactionMgr.idl>

interface ZVSServer : ZVSObject, ZVSManager
{
   // attribute ZVSLCE            LCE;  // inherited from ZVSObject
   // attribute ZVSServiceType    ST;   // inherited from ZVSManager
   readonly attribute string         Service[NameSize];
   readonly attribute string         Cell[NameSize];

   attribute ClientMgr      CM;   // pointer
   attribute CoopAgentMgr   CAM;  // pointer
   attribute ThreadMgr      TM;   // pointer
   attribute ViewMgr        VM;   // pointer
   attribute TransactionMgr TRM;  // pointer

   // Dispatch(in short Destiny, in ZVSRequest request);
   // inherited from ZVSManager
};
```

A ClientMgr object maintains a link with host managers or remote $\mathcal{ZMS}$ servers from which the requests are issued. The request is processed based on the command table

associated with ClientMgr. There are four types of client requests originated from instances of C1Client, C2Client, ParticipatingDBMS, and RemoteZVSServer respectively. The requests from a ParticipatingDBMS object is handled by a CoopAgentMgr object. Other types of requests are handled by the ClientMgr object.

```
// IDL source file: ClientMgr.idl
#include <ZVSServer.idl>

interface ClientMgr : ZVSServer
{
   attribute ZVSServer server;    // back pointer
};
```

A CoopAgentMgr object maintains a link with the cooperating agent that previously sent requests/messages to the $\mathcal{ZMS}$ server. Most of these requests/messages are originally resulted from the $\mathcal{ZMS}$ clients and the $\mathcal{ZMS}$ threads. The CoopAgentMgr object processes the requests/messages and contact the thread manager to take appropriate actions; e.g. abort the thread, commit the thread, contact the requesting client, etc.

```
// IDL source file: CoopAgentMgr.idl
#include <ZVSServer.idl>

interface CoopAgentMgr : ZVSServer
{
   attribute ZVSServer server;    // back pointer
};
```

A thread is a single sequential flow of control. The thread utility allows multiple lightweight threads to run concurrently within a single address space. In the $\mathcal{ZMS}$, we support the notion of threads to implement concurrency in a distributed system. We have discussed the semantics of the $\mathcal{ZMS}$ threads in previous sections. ThreadMgr specifies the facilities for managing a list of threads. The operation, split, handles the case when a $\mathcal{ZMS}$ thread has to be divided into two individual threads.

```
// IDL source file: ThreadMgr.idl
#include <ZVSServer.idl>
#include <ThreadList.idl>

interface ThreadMgr : ZVSServer
{
   attribute ZVSServer  server;          // back pointer
   attribute ThreadList TL;

   ReturnCode open(in ZVSLCE env, in Thread thread);
   ReturnCode split(in ZVSLCE env, in Thread thread);
   ReturnCode close(in ZVSLCE env, in Thread thread);
   ReturnCode abort(in ZVSLCE env, in Thread thread);
};
```

A ViewMgr object manages a $\mathcal{Z}$eus view repository. The $\mathcal{ZMS}$ system configuration information includes a list of $\mathcal{ZMS}$ view repositories. ViewMgr defines common operations on $\mathcal{Z}$eus views; e.g., creation, deletion, update, etc. A $\mathcal{Z}$eus view may be moved to another view repository through the migrate operation. The operation, find, can be used to look up and retrieve a $\mathcal{Z}$eus view. The ViewMgr object also handles the mappings of $\mathcal{Z}$eus view objects to CORBA objects which are stored as

view objects.

```
// IDL source file: ViewMgr.idl
#include <ZVSServer.idl>
#include <ZMSViewRepository.idl>

interface ViewMgr : ZVSServer
{
   attribute ZVSServer         server;    // back pointer
   attribute ZMSViewRepository VR;        // pointer

   ReturnCode create(in ZVSLCE env, in ZMSView view);
   ReturnCode remove(in ZVSLCE env, in ZMSView view);
   ReturnCode update(in ZVSLCE env, in ZMSView view);

   ReturnCode migrate(in ZVSLCE env,
                      inout ZMSView view,
                      in ZMSViewRepository toVR);

   ZMSView find(in ZVSLCE env, in long ID);

   ReturnCode map(in ZVSLCE env,in ZMSView view,out ZMSView CORBAObject);
};
```

The $\mathcal{Z}$eus $\mathcal{V}$iew $\mathcal{M}$echanism ($\mathcal{ZVM}$) defines how the $\mathcal{ZMS}$ requests are processed. Although the actual syntax and semantics of the $\mathcal{ZMS}$ request are still under investigation, we have proposed a two-step mapping along with a computation model that define the overall skeleton for processing the $\mathcal{ZMS}$ requests. Part of the computation model is the semantics of a $\mathcal{ZMS}$ transaction. The attribute, SR, defines the syntax rule for interpreting a $\mathcal{ZMS}$ transaction.

```
// IDL source file: TransactionMgr.idl
#include <ZVSServer.idl>
#include <ZMSTransactionList.idl>
// #include <ZMSTransaction.idl>
#include <ZMSGlobalDecl.idl>

interface TransactionMgr : ZVSServer
{
   attribute ZVSServer          server;    // back pointer
   attribute ZMSTransactionList TL;
   attribute SyntaxRule         SR;

   ReturnCode BeginTransaction(in ZVSLCE env, in ZMSTransaction tr);
   ReturnCode AbortTransaction(in ZVSLCE env, in ZMSTransaction tr);
   ReturnCode EndTransaction(in ZVSLCE env, in ZMSTransaction tr);
};
```

B.4 ZMS Client

There are four types of $\mathcal{ZMS}$ clients: C1Client, C2Client, RemoteZVSServer, and ParticipatingDBMS. Instances of C1Client, i.e., C1 $\mathcal{ZMS}$ clients, provide services for the management of views include the creation, deletion, update, browsing and installation of views in the view repository. In addition to these services, C1 $\mathcal{ZMS}$ clients also accept requests for generating IDL modules and language mappings for *Zeus* views. Since each *Zeus* view carries the information about what resources are accessible through the view, users can look up what global resources are available by browsing views via C1 $\mathcal{ZMS}$ clients. Instances of C2Client, i.e., C2 $\mathcal{ZMS}$ clients provide services for users to access global resources through views. There are several alternatives to deliver the services provided by C2 $\mathcal{ZMS}$ clients, e.g., global query language, query translation, interactive query interface, application program interface (API), and so on. Instances of RemoteZVSServer are proxy objects which are

created on behalf of a remote $\mathcal{ZMS}$ server in order to process the requests resulting from remote hosts. Instances of ParticipatingDBMS are closely integrated with participating DBMSs while providing a link to the $\mathcal{ZMS}$.

```
// IDL source file: ZVSClient.idl
#include <ZVSObject.idl>
#include <ZVSManager.idl>

interface ZVSClient : ZVSObject, ZVSManager
{
   // attribute ZVSLCE            LCE;  // inherited from ZVSObject
   // attribute ServiceType       ST;
};
```

A C1 $\mathcal{ZMS}$ client has a list of associated view repositories and an associated user interface. Both C1 and C2 $\mathcal{ZMS}$ clients are subclasses of ZVSManager. Therefore, they do not have to run on the same host as that of the host manager and they may have their own syntax for communicating with the $\mathcal{ZMS}$.

```
// IDL source file: C1Client.idl
#include <ZVSClient.idl>
#include <ZMSViewRepositoryList.idl>

interface C1Client : ZVSClient
{
   attribute ZMSViewRepositoryList VRL;

   void UserInterface();
};
```

```
// IDL source file: C2Client.idl
#include <ZVSClient.idl>

interface C2Client : ZVSClient {};
```

RemoteZVSServer is also a subclass of ZVSManager. The attribute, CmdTbl, of RemoteZVSServer is inherited from ZVSManager and defines how the proxy object of a RemoteZVSServer object interacts with the $\mathcal{ZMS}$.

```
// IDL source file: RemoteZVSServer.idl
#include <ZVSClient.idl>
#include <DirectoryServices.idl>

interface RemoteZVSServer : ZVSClient
{
   attribute string     RemoteCell[NameSize];
   attribute string     RemoteHost;
   attribute Connection link;    // Directory services
};
```

A ParticipatingDBMS object runs on the same host as that of its associated DBMS. Since ParticipatingDBMS is a subclass of ZVSManager, it inherits from ZVSManager a command interface for interacting with the $\mathcal{ZMS}$. The interactions between a ParticipatingDBMS object and DBMSs are derived through the refinement of domain frameworks.

```
// IDL source file: ParticipatingDBMS.idl
#include <ZVSClient.idl>

interface ParticipatingDBMS : ZVSClient
{
   attribute string DBMSType;
   attribute string DBMShost;
};
```

B.5 ZMS Agent

The $\mathcal{ZMS}$ agents sit between the $\mathcal{ZMS}$ server and participating DBMSs/clients. There are two types of $\mathcal{ZMS}$ agents which are subclasses of ZVSAgent; i.e., HostManager and CooperatingAgent. ZVSAgent is an abstract class.

```
// IDL source file: ZVSAgent.idl
#include <ZVSObject.idl>
#include <ZVSManager.idl>

interface ZVSAgent : ZVSObject, ZVSManager {};
```

B.5.1 ZMS Host Manager

The $\mathcal{ZMS}$ host managers are instances of HostManager. HostManager has three subclasses: ClientMgr, ServerMgr, and CoopAgentMgr. ClientMgr has one subclass: ClientMemoryMgr. A HostManager object is a multi-service network server. Incoming messages/requests are dispatched to appropriate subordinate managers for further processing.

```
// IDL source file: HostManager.idl
#include <agent.idl>
#include <HM-ClientMgr.idl>
#include <HM-ServerMgr.idl>
#include <HM-CoopAgentMgr.idl>

interface HostManager : ZVSAgent
{
   // attribute ZVSLCE LCE;  // inherited from ZVSObject
   attribute ClientMgr     CM;
   attribute ServerMgr     SM;
   attribute CoopAgentMgr  AM;
};
```

Instances of ClientMgr interact with clients, host managers, and client memory managers. Since ClientMgr is a subclass of ZVSManager, a ClientMgr object is also a network server that is able to send/receive messages/requests. The requests that are routed to a ClientMgr object may come from the ClientMgr object on another host. These requests are originated from C1 $\mathcal{ZMS}$ clients, C2 $\mathcal{ZMS}$ clients, or remote $\mathcal{ZMS}$ servers.

```
// IDL source file: HM-ClientMgr.idl
#include <HostManager.idl>

interface ClientMgr : HostManager
{
   // attribute ZVSLCE LCE;  // inherited from ZVSObject
   attribute HostManager HM; // back pointer
   attribute ClientMemoryMgr CMM;
 };
```

The client memory manager makes the retrived data available to the requesting client. The client memory manager is an instance of ClientMemoryMgr. Client-side cache management is part of the responsibilities of ClientMemoryMgr objects for handling large objects. The operation, Load, allocates a memory buffer for the data to be made available to clients and returns a pointer to that buffer.

```
// IDL source file: HM-ClientMemoryMgr.idl
#include <HM-ClientMgr.idl>

interface ClientMemoryMgr : ClientMgr
{
   attribute ClientMgr CM;   // back pointer

   void Load(in ZVSLCE env, in void dataptr);  // return a pointer
};
```

A ServerMgr object is a subordinate manager of a host manager. A ServerMgr object interacts with the $\mathcal{ZMS}$ server on behalf of a host manager. Suppose a local ClientMgr object wants to send a message to the $\mathcal{ZMS}$ server. The message is sent to a local host manager first. The host manager dispatches the message to a local ServerMgr object. The ServerMgr object then establishes a connection and sends the message to the $\mathcal{ZMS}$ server.

```
// IDL source file: HM-ServerMgr.idl
#include <HostManager.idl>

interface ServerMgr : HostManager
{
```

```
    attribute HostManager HM; // back pointer
};
```

A CoopAgentMgr object is a subordinate manager of a host manager. A CoopAgentMgr object interacts with cooperating agents on behalf of a host manager. Suppose a CoopAgentMgr object wants to send a message to a ParticipatingDBMS object, the message is sent to a local host manager first. The host manager then dispatches the message to a cooperating agent which relays the message to a DBMSAgent object. The DBMSAgent object may create a portal based on the message. The portal communicates with a ParticipatingDBMS object and sends the response back to the requesting CoopAgentMgr object.

```
// IDL source file: HM-CoopAgentMgr.idl
#include <HostManager.idl>

interface CoopAgentMgr : HostManager
{
   attribute HostManager HM; // back pointer
};
```

B.5.2 ZMS Cooperating Agent

The $\mathcal{ZMS}$ Cooperating Agent, CooperatingAgent, is the root of the database framework. CooperatingAgent has three subclasses: DBMSAgent, ServerAgent, and ClientAgent. Instances of CooperatingAgent are multi-service network servers that send/receive messages and dispatch requests to their subordinate managers. Each

cooperating agent maintains a list of associated DBMS agents.

```
// IDL source file: CoopAgent.idl
#include <ZVSAgent.idl>
#include <CA-ServerAgent.idl>
#include <CA-ClientAgent.idl>
#include <DBMSAgentList.idl>

interface CooperatingAgent : ZVSAgent
{
   attribute ServerAgent   SA;       // pointer
   attribute ClientAgent   CA;       // pointer
   attribute DBMSAgentList DBMSs;
};
```

A DBMS agent is specific to a DBMS. A DBMS agent is an instance of DBMSAgent that defines the associated templates and the operations for manipulating templates and portals. Notice that we may create a user interface for the management of DBMS templates. Such a user interface can be created as a C2 $\mathcal{ZMS}$ client.

```
// IDL source file: DBMSAgent.idl
#include <coopAgent.idl>
// #include <Template.idl>
#include <TemplateList.idl>
#include <Portal.idl>
#include <ZMSView.idl>
interface DBMSAgent : CooperatingAgent
{
   attribute CooperatingAgent CA;            // back pointer
   attribute TemplateList     templates;

   ReturnCode CreateTemplate(in ZVSLCE env, inout Template template);
```

```
    ReturnCode DeleteTemplate(in ZVSLCE env, inout Template template);
    ReturnCode UpdateTemplate(in ZVSLCE env, inout Template template);

    ReturnCode SaveTemplate(in ZVSLCE env, inout Template template);
    ReturnCode RestoreTemplate(in ZVSLCE env, inout Template template);

    ReturnCode CreatePortal(in    ZVSLCE   env,
                            inout Template template,
                            in    ZMSView  view,
                            out   Portal   portal);
};
```

A DBMS may provide multiple interfaces by defining multiple templates. A template models a generic interface that can be used to automate the generation of the DBMS access routines, i.e. portals, based on a user-defined view. The result is a customized interface to the DBMS tailored to the requirements of a specific application. The operation, TemplateToPortal, generates a customized interface for a *Zeus* view. The generated interface can be used to create portals.

```
// IDL source file: Template.idl
#include <DBMSAgent.idl>
#include <Portal.idl>

interface Template : DBMSAgent
{
    attribute DBMSAgent agent;              // back pointer
    attribute DBMSType  DT;
    attribute void      specification;
    attribute void      TemplateStore;      // pointer

    ReturnCode TemplateToPortal(in  ZVSLCE env,
                                in  ZMSView view,
                                out Portal portal);
};
```

A portal is a runtime entity that executes the instructions for sending requests to and receiving data from a DBMS. The application semantics embedded in a portal is derived from the *Zeus* views involved in a request. The operation, **run**, implements the runtime application semantics.

```
// IDL source file: Portal.idl
#include <DBMSAgent.idl>
#include <ParticipatingDBMS.idl>
#include <Template.idl>

interface Portal : DBMSAgent
{
   attribute DBMSAgent         agent;     // back pointer
   attribute Template          template;  // back pointer
   attribute ParticipatingDBMS contact;

   ReturnCode run(in ZVSLCE env);
};
```

A ServerAgent object interacts with a *ZMS* server on behalf of a cooperating agent.

```
// IDL source file: CA-ServerAgent.idl
#include <CoopAgent.idl>

interface ServerAgent : Cooperating Agent
{
   attribute CooperatingAgent CA;   // back pointer
};
```

A ClientAgent object interacts with a $\mathcal{ZMS}$ client on behalf of a cooperating agent.

```
// IDL source file: CA-ClientAgent.idl
#include <CoopAgent.idl>

interface ClientAgent
{
   attribute CooperatingAgent CA;   // back pointer
};
```

B.6 ZMS Environment

The $\mathcal{ZMS}$ environment refers to the computing environment of a certain $\mathcal{ZMS}$ component. The class, ZVSLCE, is used to model the $\mathcal{ZMS}$ environment. Instances of ZVSLCE provide links to access the services of the hosting computing environment. We have identified four major components of the $\mathcal{ZMS}$ environment: network services, operating system services, object request broker, and persistent object store.

```
// IDL source file: ZVSLCE.idl
#include <ZMSGlobalDecl.idl>

interface ZVSLCE
{
   attribute short                  ID;  // identifier
   attribute NetworkServices        NS;  // pointer
   attribute OperatingSystemServices OS;  // pointer
   attribute ObjectRequestBroker    ORB; // pointer
   attribute PersistentObjectStore  POS; // pointer
   attribute HostType               HT;
```

```
    attribute string                HostName[NameSize];
};
```

B.6.1 Network Services

There are three major network services: name service, object transfer, and remote method invocation. The name service provides two levels of name services: the name service in the context of the $\mathcal{ZMS}$, and the name service for common network services. The object transfer and remote method invocation support the object-level network services required by the $\mathcal{ZMS}$.

```
// IDL source file: NetworkServices.idl
#include <ZVSLCE.idl>

interface NetworkServices : ZVSLCE
{
   attribute ZVSLCE            env;     // back pointer
   attribute NameService       NS;
   attribute ObjectTransferMgr OTM;
   attribute RMIMgr            RMIM;
};
```

We group common network services in NameService. NameService defines the attributes that point to common network services. These network services may be provided as application program interfaces (APIs). The specifications of DirectoryServices, SecurityServices, TimeServices and FileServices are not included in our current specification. We have two alternatives to determine the specifications of these common network services. First of all, we may continue to abstract commonalities from

different implementations of these network services. Secondly, we may abstract specifications from a specific implementation. The operation, **search**, can be used to find an entity in a $\mathcal{ZMS}$ by giving a name following the $\mathcal{ZMS}$ naming convention.

```
// IDL source file: NameService.idl
#include <NetworkServices.idl>
#include <DirectoryServices.idl>
#include <SecurityServices.idl>
#include <TimeServices.idl>
#include <FileServices.idl>
#include <ZMSGlobalDecl.idl>
#include <ZMSystem.idl>

 interface NameService : NetworkServices
 {
    attribute Cell              domain;
    attribute DirectoryServices DS;          // pointer
    attribute SecurityServices  SS;          // pointer
    attribute TimeServices      TS;          // pointer
    attribute FileServices      FS;          // pointer

    void search(in ZMSystem system, in string name);
 };
```

ObjectTransferMgr specifies the object transfer interface. The operations, **send** and **receive**, support the transfer of typed object. The operation, **pipe**, implements a pipe mechanism that supports reliable transfers of large objects. The pipe mechanism can be used when the transfer is required for large quantities of data, data of unknown size that cannot fit in memory, or data incrementally produced and not in memory all at once.

```
// IDL source file: ObjectTransferMgr.idl
#include <ZVSLCE.idl>
#include <NetworkServices.idl>
#include <Pipe.idl>

interface ObjectTransferMgr : NetworkServices
{
   attribute ZVSLCE env;       // back pointer

   ReturnCode send(in ZVSLCE env, inout void object);
   ReturnCode receive(in ZVSLCE, in void object);
   ReturnCode CreatePipe(in ZVSLCE, inout void object, inout Pipe pipe);
};
```

There are two types of pipes: input pipe and output pipe. An input pipe supports the transfer of data from a client to a server. An output pipe supports the transfer of data from a server to a client. The operation, **pull**, is used by an input pipe. The operation, **push**, is used by an output pipe. The operation, **allocate**, allocates the memory buffer for the chunk of data transferred. The attribute, PS, defines the pipe state that is specific and local to either the client or the server side.

```
// IDL source file: Pipe.idl
#include <ZMSGlobalDecl.idl>

interface Pipe
{
   readonly attribute void      PS;          // pipe state
   readonly attribute PipeType  PT;

   ReturnCode pull(void object);
   ReturnCode push(void object);
   void allocate();
};
```

The remote method invocation service supports the invocation of methods associated with the objects on remote hosts. A remote method must be be registered with the local $\mathcal{ZMS}$ via the **register** operation. The operation, **open**, creates a connection with a remote agent that will execute the method called by the following **invoke** operation.

```
// IDL source file: RMIMgr.idl
#include <ZVSLCE.idl>
#include <NetworkServices.idl>
#include <ZMSGlobalDecl.idl>

interface RMIMgr : NetworkServices
{
   attribute ZVSLCE env;       // back pointer

   ReturnCode register(in ZVSLCE env, in RemoteMethod RM);
   ReturnCode open(in ZVSLCE env, inout void RemoteAgent);
   ReturnCode invoke(in ZVSLCE env, in RemoteMethod RM);
};
```

B.6.2 Operating System Services

The class, **OperatingSystemServices**, specifies the operating system services required by the $\mathcal{ZMS}$. How these services are implemented is not a major concern in the development of the $\mathcal{ZMS}$. We classify high-level operating system services in four categories: Process Management, Memory Management, I/O Subsystem, and Interprocess Communication. Process Management includes the control and scheduling of processes. Memory Management includes the allocation of main memory for

running processes, or the allocation of secondary memory, e.g. file systems. I/O Subsystem manages controlled access to peripheral devices. Interprocess Communication provides mechanisms for arbitrary processes to exchange data and synchronize execution. The above services are not included in our specifications. Operating system services are part of the environment framework. A subclass of OperatingSystemServices that corresponds to a specific operating system on a certain platform can be derived through refinement and specialization of OperatingSystemServices. The thread utility should be included as part of ProcessMgmt.

```
// IDL source file: OperatingSystemServices.idl
#include <ZVSLCE.idl>
#include <ProcessMgmt.idl>
#include <IOSubsystem.idl>
#include <InterProcessComm.idl>

interface OperatingSystemServices : ZVSLCE
{
   attribute ProcessMgmt      PM;      // pointer
   attribute MemoryMgmt       MM;
   attribute IOSubsystem      IOS;
   attribute InterPorcessComm IPC;
};
```

B.6.3 Object Request Broker

The design of the $\mathcal{ZMS}$ requires that the hosts of all the $\mathcal{ZMS}$ components must be running an object request broker. Since there are different implementations of CORBA, we provide a class, ObjectRequestBroker, to describe the common features of object request brokers. The interface object, Interoperability, defines interoperability issues at the levels of communications, binding, interface references, and

interception[131]. We assume that ORB.idl, InterfaceRepository.idl, and Interoperability.idl are provided as part of the object request broker.

```
// IDL source file: ObjectRequestBroker.idl
#include <ORB.idl>
#include <InterfaceRepository.idl>
#include <Interoperability.idl>

interface ObjectRequestBroker : ZVSLCE
{
   attribute ORB                 orb;
   attribute InterfaceRepository *IR;
   attribute Interoperability    INTOPR;
};
```

B.6.4 Persistent Object Store

Persistent object stores are used in the $\mathcal{ZMS}$ to support the save/restore of objects of any type or complexity. PersistentObjectStore has its own command syntax defined in the attribute, CmdTbl. We use void as the type for parameters in several occasions where there are two other possible alternatives. First of all, we may have used parameterized type if it were supported in CORBA IDL. Secondly, we may provide an exhaust list of signatures for the same operation such that all possible types are covered for a specific parameter although this is not a good solution.

```
// IDL source file: zvspos.idl
#include <ZVSLCE.idl>
#include <ZMSGlobalDecl.idl>
```

```
interface PersistentObjectStore : ZVSLCE
{
   // attribute ZVSLCE LCE;        inherited from ZVSObject

   readonly attribute CommandTableEntry CmdTbl;

   ReturnCode startup();
   ReturnCode shutdown();
   ReturnCode Parser(in ZVSLCE env, in ZVSRequest request);

   ReturnCode POSSetObjectID(in ZVSLCE env, void objectID);
   ReturnCode POSStoreObject(in ZVSLCE env, inout void dataptr);
   ReturnCode POSRestoreObject(in ZVSLCE env, inout void path);
   ReturnCode POSFindObject(in ZVSLCE env, void objectID);
   ReturnCode CheckError(in ZVSLCE env);
};
```

B.7 Miscellany

B.7.1 Global Declarations

The interface, ZMSGlobalDecl, includes global declarations used by other IDL source files. ZVSServiceType defines the names of all $\mathcal{ZMS}$ service types provided. ReturnCode defines the $\mathcal{ZMS}$-specific messages that may initiate other event handling routines.

```
// IDL source file: ZMSGlobalDecl.idl

interface ZMSGlobalDecl
{
   const short NameSize = 64;

   enum ZVSServiceType {Server,
                        C1Client, C2Client,
```

```
                    RemoteServer, ParticipatingDBMS,
                    HostManager,
                    CooperatingAgent};

enum ViewType {LocalInterface, BaseView, View, Subscribed};

enum ReturnCode {
       ServerBusy,                  // No server available
       ConnectionLost               // Lost connection
     };

enum TransactionStatusType {Succeed, Fail, Aborted};

enum ThreadStatus {
       Active,              // The thread is still active.
       Completed,           // The thread has been completed.
       Aborted              // The thread is aborted.
     };

enum ThreadElementType {
       C1,                  //
       C2,                  //
       RemoteServer,        //
       ParticipatingDBMS,   //
     };

enum HostType {
       DEC5000,
       AS400
     };

enum PipeType {
       input,
       output
     };

struct CommandTableEntry {
         string    CommandName;
         short     CommandFunction;  // pointer
         string    Help;
         string    Syntax;
     };
```

```
    struct RemoteMethod {
             void      object;
             string    method;
           };

    typedef time          long;
    typedef TypeRelations string;
    typedef SyntaxRule    string;
    typedef Cell          string;
};
```

B.7.2 ZMS Views & View Repository

There are four types of $\mathcal{ZMS}$ views: local interface, base view, view, and subscribed view. We have provided formal definitions for these $\mathcal{ZMS}$ views in Chapter 3. Each view has an identifier assigned by the $\mathcal{ZMS}$. At runtime, a view is addressable via a pointer. The attribute, DataToRetrieve, describes the data to be retrieved from participating DBMSs. The attribute, Structure, describes the structure that will host the retrieved data. The attribute, MetaInfo, contains the meta information of views.

```
// IDL source file: ZMSView.idl
#include <ZMSGlobalDecl.idl>
#include <ZMSViewRepository.idl>

interface ZMSView
{
    attribute long     ID;
    attribute string   name[NameSize];
    attribute ViewType VT;       // local interface, base view, view,
                                 // or subscribed view
    attribute string   DataToRetrieve;
```

```
                                // describe the data to be retrieved
    attribute short    DTRsize;  // size of DataToRetrieve
    attribute string   Structure;// describe the structure to host data
    attribute short    SSize;    // size of Structure
    attribute string   MetaInfo; // Meta information; e.g., manual
    attribute short    MISize;   // size of MetaInfo

    attribute TypeRelations TR;  // pointer

    attribute ViewRepository VR; // back pointer

    ReturnCode create(in ZMSView view);
    ReturnCode delete(in ZMSView view);
    ReturnCode update(in ZMSView view);

};
```

The $\mathcal{ZMS}$ view repositories stores $\mathcal{Z}eus$ views in a persistent object store. The associated persistent object store provides object storage and index structures. ViewRepository defines the interface for the management and manipulation of $\mathcal{Z}eus$ views. The type, SyntaxRule, hosts the abstract syntax used for interpreting the abstract syntactic descriptions of type relations, structures, etc.

```
// IDL source file: ZMSViewRepository.idl
#include <ZMSView.idl>
#include <ZVSLCE.idl>
#include <PersistentObjectStore.idl>
#include <ZMSGlobalDecl.idl>

interface ViewRepository
{
    attribute ZVSLCE                env;
    attribute short                 RepositoryID;
    attribute PersistentObjectStore POS;          // pointer
    attribute short                 TRSize;
```

```
    attribute Index                 VRIndex;      // accessed via POS

    attribute SyntaxRule            TypeRelations;
    attribute SyntaxRule            DataAccess;
    attribute SyntaxRule            Structure;

    ReturnCode find(in ZVSLCE env, inout ZMSView view);
    ReturnCode create(in ZVSLCE env, in ZMSView view);
    ReturnCode delete(in ZVSLCE env, in ZMSView view);
    ReturnCode update(in ZVSLCE env, in ZMSView view);
    ReturnCode publish(in ZVSLCE env, in ZMSView view);
    ReturnCode subscribe(in ZVSLCE env, in string name);
};
```

B.7.3 Thread, Request & Transaction

A thread is the basic unit of computation within the $\mathcal{ZMS}$ Server. A thread is spawned upon a request. The semantics of the request is translated to the logic carried by the thread. A thread may spawn multiple thread elements and coordinates the completion for each of its thread element. Each thread element has an attribute that links to the corresponding system thread supported and implemented at the hosting operating system level.

```
// IDL source file: thread.idl
#include <ZMSGlobalDecl.idl>
#include <TimeServices.idl>
#include <ThreadMgr.idl>
#include <ThreadElementList.idl>

interface Thread
{
    attribute ThreadStatus status;
    attribute string       origin;   // Created on behalf of the origin
```

```
   attribute ThreadMgr    TM;       // back pointer
   attribute Time         CreationTime;
   attribute Time         CloseTime;// Or the time the thread is aborted.

   readonly attribute ThreadElementList TEs;    // Thread Elements

   ReturnCode CreateThread(in ZVSLCE env, in Thread thread);
   ReturnCode DeleteThread(in ZVSLCE env, in Thread thread);

   ReturnCode CloseThread(in ZVSLCE env, in Thread thread);
   ReturnCode AbortThread(in ZVSLCE env, in Thread thread);
};
```

```
// IDL source file: ThreadElement.idl
#include <ZMSGlobalDecl.idl>
#include <TimeServices.idl>
#include <thread.idl>
#include <ProcessMgmt.idl>

interface ThreadElement : Thread
{
   attribute ZVSLCE             env;
   attribute ThreadElementType  TET;
   attribute Thread             parent;
   attribute Thread             link;// point to the parent before split
   attribute Time               CreationTime;
   attribute Time               CloseTime;
   attribute SystemThread       ST;  // pointer

   ReturnCode CreateThreadElement(in ZVSLCE env, in ThreadElement TE);
   ReturnCode DeleteThreadElement(in ZVSLCE env, in ThreadElement TE);

   ReturnCode CloseThreadElement(in ZVSLCE env, in ThreadElement TE);
   ReturnCode AbortThreadElement(in ZVSLCE env, in ThreadElement TE);
};
```

All the requests and exchanged messages in the $\mathcal{ZMS}$ are modeled by ZVSRequest. How these requests and messages are handled depends on the involved network servers

and clients. RequestSyntax identifies the command table in case the target network server is multi-lingual. The attributes, requestText and TextSize, contain the text and size of the request, respectively.

```
// IDL source file: ZVSRequest.idl
#include <ZMSGlobalDecl.idl>

interface ZVSRequest
{
   attribute CommandTableEntry RequestSyntax; // pointer
   attribute void              source;    // Where the request is from
   attribute void              target;    // Where the request is going
   attribute string            requestText;
   attribute long              TextSize;
};
```

Some $\mathcal{ZMS}$ requests may result in transactions. The $\mathcal{ZMS}$ transactions are modeled by ZMStransaction. A $\mathcal{ZMS}$ transaction has a identifier assigned by the $\mathcal{ZMS}$. A transaction may result in a list of $\mathcal{ZMS}$ threads. The attribute, TL, points to a list of threads associated with the transaction.

```
// IDL source file: ZVStransaction.idl
#include <ZVSRequest.idl>
#include <ThreadList.idl>
#include <TransactionMgr.idl>

interface ZMSTransaction
{
    attribute long          ID;
    attribute ZVSRequest    request;      // pointer
    attribute ThreadList    TL;           // pointer
```

```
    attribute TransactionMgr TM;          // back pointer

    ReturnCode create(in ZMSTransaction transaction);
    ReturnCode commit(in ZMSTransaction transaction);
    ReturnCode abort(in ZMSTransaction transaction);
};
```

B.7.4 Utilities

We provide two generic classes, List and ListElement, for modeling linked lists of the $\mathcal{ZMS}$ components. Different types of lists are created as subclasses of List.

```
// IDL source file: List.idl
#include <ListElement.idl>

interface List
{
   attribute long count;
   attribute ListElement head;          // pointer
   attribute ListElement tail;          // pointer

   ListElement head();
   ListElement tail();
   ListElement tally();
   ReturnCode  append(in ListElement le);
   ReturnCode  prepend(in ListElement le);
   ReturnCode  remove(in ListElement le);
   ReturnCode  has(in ListElement le);
};
```

```
// IDL source file: ListElement.idl>

interface ListElement
```

```
{
   attribute void        dataptr;
   attribute ListElement next;
   attribute ListElement prev;
};
```